

L'INTERPRETEUR **BASIC 1.0** **des** **TO 7-70 et TO 9**

le
comprendre

le
compléter

l'
améliorer

CHEZ LE MEME EDITEUR

- AUBERT et SCHOMBERG** *Pratiquez l'intelligence artificielle* 144 p. ; 1985. (coll. Micro-ordinateurs).
- BERNARD** - *L'assembleur facile du 6809* 168 p. ; 1985, (coll. Micro-ordinateurs).
- BUI MINH DUC** *Programmation en assembleur 6809*. 388 p. ; 1985.
- DARDANNE** *Microprocesseur 6809. Ses périphériques et le processeur graphique* 9365.66 304 p. ; 1984.
- DELAHAYE** - *Dessins géométriques et artistiques avec votre micro-ordinateur* 256 p. ; 1985.
- GUILLON** *La conduite du T07-70* 208 p.; 1985, (coll. Micro-ordinateurs).
- KRIEGER** - *Programmes pédagogiques sur T07, T07-70, MO5* 168 p. ; 1985, (coll. E.A.O.).
- KRUTCH** - *Expériences d'intelligence artificielle en Basic* - 128 p. ; 1984, (coll. Micro-ordinateurs).
- VANRYB et POLITIS** *MSX. Basic MSX et MSX-DOS* 212 p.; 1985.
- VEY** - *Apprentissage et utilisation du bus IEEE 488/CEI 625* - 232 p. ; 1985.
- WANNER** - *Aller plus loin en Basic T07* - 312 p. ; 1985, (coll. Microplus).

Introduction

Il est peu habituel de se soucier de la manière dont un programme BASIC est exécuté par un ordinateur.

Ce dernier n'est pourtant capable d'exécuter directement que des instructions tout à fait élémentaires, beaucoup moins complexes qu'un FOR ou IF ou autres BOXF...

Nous nous intéressons donc dans cet ouvrage à l'interpréteur, c'est-à-dire au programme chargé de décoder et exécuter des instructions écrites en BASIC.

Nous avons choisi pour cela la très répandue version 5.0 de la société MICROSOFT, écrite ici pour les deux micro-ordinateurs THOMSON T07 et T07-70 et leur microprocesseur 6809.

Après avoir présenté dans une première partie les deux T07 et étudié en détail la programmation du 6809, la deuxième partie permet de comprendre le fonctionnement de l'interpréteur, commun aux deux T07.

Les deux autres parties de cet ouvrage sont ensuite consacrées aux applications, consistant d'abord à modifier et compléter le BASIC d'origine, puis à améliorer ses performances.

Ce livre se veut à la portée aussi bien du programmeur "avancé" qu'à celle du débutant, qui pourra dans un premier temps se consacrer aux applications et découvrir ensuite la programmation en langage machine.

Il poursuit un triple but:

- *But "pédagogique" d'abord, sans qu'il soit question de théorie, avec la présentation détaillée du 6809, la méthode générale de décodage d'un interpréteur quelconque, la présentation de quelques notions importantes en général méconnues du programmeur BASIC comme celles de portée d'une variable ou de récursivité et enfin la découverte de ce qui se passe "derrière" les GOTO, IF ou FOR habituels.*
- *But "utilitaire" ensuite, avec la présentation de routines ou de programmes BASIC permettant de transformer l'interpréteur, tant au point de vue utilisation (nouvelles fonctions et instructions, procédures, sprites, BASIC français...) qu'à celui des performances (compilation).*
- *But "incitatif" enfin, en donnant tous les moyens de mettre en œuvre les propres idées de chacun, et toutes les méthodes permettant de comprendre et de transformer de même qu'ici l'interpréteur BASIC d'un ordinateur quelconque (MO5 en particulier).*

Table des matières

Introduction	VII
 PREMIÈRE PARTIE. — LES T07. LE 6809. LE LANGAGE MACHINE	 1
1. Les T07	3
Caractéristiques générales	3
Le BASIC	4
Organisation de la mémoire	5
2. Le moniteur	7
Gestion de l'écran : routine PUTCS	8
Le clavier	10
Affichages graphiques	11
Autres routines-registres du moniteur	13
3. Le 6809	14
Système binaire-codage hexadécimal	14
Les registres internes	17
	IX

Modes d'adressage	20
Jeu d'instructions	24
4. Le langage machine	29
Implantation d'un programme	30
Exécution-passage de paramètres avec le BASIC	32
Désassemblage	33
 DEUXIÈME PARTIE. — L'INTERPRÉTEUR BASIC MICROSOFT	 43
1. Comprendre l'exécution d'un programme BASIC	45
Implantation et codage d'un programme	46
La table et les codes des mots réservés	47
La table des adresses d'instructions	50
Le traitement des instructions	52
Boucle d'exécution d'un programme	55
2. Le traitement des variables	57
Représentation des variables	57
Représentation des tableaux	59
Gestion de la mémoire	59
Recherche d'une variable ou d'un tableau	60
Traitement d'une expression	62
Traitement des fonctions BASIC	65
Traitement des opérateurs	66
3. Étude de quelques instructions BASIC	69
Instructions de branchement	69
Instruction de test	72
Instruction FOR...NEXT	72
Les instructions graphiques	74
Les autres instructions	77
4. Méthode pratique de décodage d'un interpréteur	78
 TROISIÈME PARTIE. — MODIFIER ET COMPLÉTER LE BASIC	 81
1. Créer et utiliser un BASIC français	83
Création du nouveau vocabulaire	83
Initialisation-utilisation	85
Table de traduction	88

2. Créer de nouvelles fonctions	89
Principe de la méthode	89
Exemples d'applications	90
Création des nouvelles fonctions	92
Utilisation du programme de création	94
3. Boucle WHILE...WEND	97
Principe de création d'une boucle WHILE	98
Les routines-initialisation	99
Exemple d'utilisation	100
4. Création et utilisation de nouvelles instructions	102
Principe de la méthode	102
Mise en œuvre	103
Création des nouvelles tables	105
5. Incrémentation et permutation de variables	108
Instruction INC	108
Instruction SWAP	110
6. Procédures-variables locales	114
Principe de la méthode-passage des arguments	115
Instruction CALL	117
Instruction PROC	121
Instruction PROCEND	123
Applications simples	125
Application aux jeux	129
Améliorations possibles	132
7. Les sprites	133
Animation graphique classique	134
Instruction SPRITE	134
Application aux jeux d'action	142
8. Résumé de la troisième partie	145
 QUATRIÈME PARTIE. — AMÉLIORER LES PERFORMANCES	 149
1. Modification du fonctionnement du clavier	151
Suppression de l'action du clavier	151
Jeux à 1 ou 2 joueurs par le clavier	153

2. Recettes classiques d'amélioration des performances	157
3. Compilation interactive des adresses de variables et des constantes	159
La méthode	160
Compilation d'une adresse	162
Modification du traitement des opérandes	165
Mise en œuvre-résultats	166
Le problème des tableaux	169
4. Compilation des adresses de branchement	171
La méthode	171
Traitement de GOTO et GOSUB	173
Traitement de l'instruction ON	174
Mise en œuvre-exemple	175
5. Application à d'autres ordinateurs	177
Cas où il existe des vecteurs en mémoire vive	177
Cas où le BASIC est entièrement figé en ROM	178
6. Résumé de la 4^e partie	180
 CINQUIÈME PARTIE. – LE T09 - LE DOS BASIC	183
1. Le T09 – Ses logiciels	184
Présentation matérielle.	185
Les logiciels ; leur étude	185
Transfert sur disquette des programmes T07-70	187
2. Le DOS BASIC 1.0	188
Implantation et codage d'un programme	188
Tables des nouvelles instructions et fonctions	189
Exécution des nouvelles instructions et fonctions ; tables des adresses	190
3. Création des nouvelles instructions – Compilation.	193
Fonctions de conversion radiants-degrés.	193
Création de nouvelles instructions.	194
Adaptation des routines	195
Compilation	199

ANNEXES

1. Routines du moniteur	201
2. Principales adresses du moniteur	203
3. Instructions du 6809	207
4. Adressages indexés et relatifs-conversions décimal/hexa ..	211
5. Code ASCII	215
6. Instructions BASIC	217
7. Fonctions BASIC	219
8. Opérateurs BASIC	221
9. Principales adresses du BASIC	223
10. Principales routines du BASIC	225

Première partie

LES TO7. LE 6809 LE LANGAGE MACHINE

Nous décrivons ici les principales caractéristiques des deux micro-ordinateurs TO7 de THOMSON, ainsi que les éléments du moniteur et du BASIC que nous utiliserons plus tard dans la suite de ce livre ; les différences entre TO7 et TO7-70 sont particulièrement développées.

Nous ne détaillons par contre pas les instructions du BASIC, partiellement décrites dans le livre d'initiation fourni avec les TO7, ou plus précisément dans divers ouvrages dont le manuel de référence.

1

Les TO7

1. Caractéristiques générales

Les deux ordinateurs TO7 et TO7-7Ø, apparus respectivement fin 82 et mi 84, sont parfaitement représentatifs de ce que l'on peut attendre aujourd'hui de l'informatique "familiale".

Nous citerons par exemple :

- Affichage en couleur : 8 pour le TO7, 16 pour le TO7-7Ø qui permet 8 couleurs supplémentaires.
- Graphisme très fin de 32Ø × 2ØØ points, entièrement mixable avec du texte.
- Présence d'un crayon optique permettant de dessiner sur l'écran, de choisir entre différentes options (menu), etc...
- Générateur de sons sur 5 octaves.

- 32 K octets de mémoire vive directement accessibles par le BASIC (plus 16 K commutables, soit 48 K en tout) pour la version de base du TO7-70 (extensibles à 112 K), contre 8 K (extensibles à 24 K) pour le TO7.
- BASIC Microsoft extrêmement complet, utilisant parfaitement la puissance du très moderne microprocesseur 6809.
- Cartouches de programmes en mémoire morte (16 K), permettant de changer instantanément d'application ou de langage (BASIC, LOGO, assembleur...).
- Interfaces en option pour le TO7-70 permettant la digitalisation d'images, l'incrustation d'images vidéo et l'accès au réseau Télétel.

II. Le BASIC

Le BASIC est actuellement commun aux deux TO7 ; il possède toutes les fonctions classiques du BASIC Microsoft dans ses versions les plus récentes (version 5.0) :

- Test IF...THEN...ELSE.
- Quatre types de variables, dont la double précision, les noms pouvant avoir 16 caractères significatifs.
- Instructions graphiques très souples : LINE, BOX, COLOR, SCREEN, etc...
- Instructions de gestion du stylo optique.
- Éditeur "plein écran".
- Traitement des erreurs : ON...ERROR..., ERR, ERL, RESUME.
- Opérations sur les fichiers de données.

Ce BASIC très complet est aussi très rapide ; il peut être encore amélioré avec l'extension disque, qui apporte de nouvelles instructions comme RENUM, PAINT, DRAW, CIRCLE, etc...

III. Organisation de la mémoire

1. Organisation générale

Elle est la même pour les deux TO7.

Adresses	Contenu
\$0000 à \$3FFF (0 à 16383)	Cartouche ROM (interpréteur BASIC) ; 16 K octets
\$4000 à \$5F3F (16384 à 24383)	Mémoire écran ; 2 fois 8 K octets situés à la même adresse
\$6000 à \$xxxx (à partir de 24576)	Mémoire RAM utilisateur ; premiers 1,5 K réservés au moniteur et au BASIC
\$E000 à \$E7BF (57344 à 59327)	Pour le DOS ; 1,9 K
\$E7C0 à \$E7E7 (59328 à 59367)	Adresses d'entrée-sortie
\$E800 à \$FFFF (59392 à 65535)	Moniteur ; 6 K octets

REMARQUE: Un nombre précédé de \$ désigne une *adresse* exprimée en hexadécimal (voir chapitre III et annexes).

Un nombre précédé de &H désignera dans toute la suite une *valeur* hexadécimale.

Sur le TO7 de base, la zone mémoire utilisateur se termine en \$7FFF(32767) ; l'extension 16 K porte l'adresse de fin à \$BFFF(49151), soient 24 K octets, dont 22,5 disponibles pour les programmes BASIC.

2. Accès aux différentes banques de mémoire du TO7-70

Sur le TO7-70, la mémoire utilisateur va jusqu'en \$DFFF(57343), soient 32 K octets (dont 30,5 disponibles pour les programmes BASIC).

La zone située entre ~~\$A000~~(40960) et \$DFFF comprend en fait 2 banques de 16 K octets commutables par logiciel ; l'extension mémoire porte ce nombre à 6.

Ces différentes banques pourront par exemple être utilisées comme mémoires auxiliaires, au temps d'accès beaucoup plus rapide que celui d'une disquette, ou encore comme deuxième page d'affichage (à envoyer une fois "remplie" dans la mémoire écran, pour créer des animations), etc...

Dans tous les cas, une seule banque est accessible à un instant donné, mais on peut très facilement les commuter en intervenant sur le PIA système 6821.

On utilisera par exemple pour cela le sous-programme en langage machine suivant, que l'on appellera avec la valeur V convenable placée dans le registre B (voir plus loin):

```
COMMUT  LDU # $E7CB
        LDA  , U
        ANDA # $FB
        STA  , U
        STB  - 2, U
        ORA  # $04
        STA  , U
        RTS
```

A → reg. de contrôle
V → reg. de données

La valeur V=15 correspond à la 1^{re} banque, la valeur 23 à la seconde, 231 à la 3^e, 103 à la 4^e, 167 à la 5^e et 39 à la 6^e.

2

Le moniteur

Nous décrivons ici les principales routines (sous-programmes) utilisées par l'interpréteur BASIC, ou bien sûr par des programmes utilisateurs écrits en langage machine et devant effectuer des entrée-sortie ; les routines correspondantes seront alors appelées par une instruction JSR.

Bien que les moniteurs des deux TO7 ne soient pas identiques, les adresses des routines sont exactement les mêmes, puisque situées entre \$E800 et \$E833 où on trouve les branchements aux routines proprement dites.

Tous les paramètres utilisés par celles-ci sont eux aussi entièrement compatibles, ce qui explique que les deux TO7 fonctionnent avec le même BASIC.

Celui-ci n'exploite toutefois pas certaines nouvelles caractéristiques du TO7-70 (16 couleurs, incrustation d'images vidéo) ; nous indiquons donc ici comment exploiter ces possibilités.

1. Gestion de l'écran : routine PUTC\$

L'affichage est de 25 lignes de 40 caractères en mode alphanumérique, et de 200 lignes de 320 points (groupés par 8) en mode graphique.

On trouvera dans la seconde partie (chapitre III) la description détaillée de la mémoire écran, constituée de 2 blocs distincts situés à la même adresse, et sélectionnés par la valeur du bit 0 du registre de données du PIA système 6846 (adresse \$E7C3).

1. Rôle de la routine-utilisation

La routine PUTC\$, située en \$E803, permet d'effectuer l'affichage de tous les caractères alphanumériques, y compris les accents, les caractères semi-graphiques et les caractères utilisateurs ; tous les mouvements du curseur sont possibles.

Elle permet aussi de gérer tous les attributs d'écran : positionnement de la fenêtre, couleur de la forme, du fond et du tour, taille des caractères, type du défilement (scrolling), masquage, mise en mode incrustation pour le TO7-70.

Cette routine est utilisée par les instructions BASIC suivantes :

- PRINT : écriture de tous les caractères alphanumériques
- LOCATE : positionnement du curseur, qui peut-être rendu invisible
- CONSOLE : définition de la fenêtre, type du défilement (avec ou sans couleur, normal, doux ou en mode page)
- SCREEN : couleurs de la forme, du fond ou du tour, qui peuvent être inversées
- ATTRB : taille des caractères, qui peuvent être masqués
- UNMASK : démasquage des caractères masqués.

Nous ne détaillerons donc pas l'emploi de la routine pour toutes les opérations ci-dessus, effectuées plus commodément à partir du BASIC.

2. Cas du TO7-7Ø

Le BASIC ne permet pas pour le TO7-7Ø l'accès aux couleurs pastels et la mise en mode incrustation.

Pour colorer la forme, le fond ou le tour en une couleur pastel, il faudra donc exécuter la séquence suivante, écrite en langage machine (voir chapitre IV) :

COLOR	LDB # \$1B	
	JSR \$E8Ø3	Séquence "d'échappement"
	LDB # Valeur	
	JSR \$E8Ø3	2 ^e appel

"Valeur" comprise entre :

- &H7Ø et &H77 : modification de la couleur de la forme
- &H78 et &H7F : modification de la couleur du fond
- &H8Ø et &H87 : modification de la couleur du tour de l'écran.

La valeur exacte à transmettre à PUTC\$ sera obtenue en ajoutant à &H7Ø ou &H78 ou &H8Ø un des nombres suivants :

- Ø → gris
- 1 → rose
- 2 → vert clair
- 3 → sable
- 4 → bleu clair
- 5 → parme
- 6 → bleu ciel
- 7 → orange

REMARQUE : les valeurs de &H4Ø à 47, &H5Ø à 57 et &H6Ø à 67 provoqueront la modification des mêmes éléments dans une couleur normale, et ce pour les deux TO7.

Enfin, le TO7-7Ø sera mis en mode incrustation (si l'extension est présente) par la séquence :

INCRUS	LDB # \$1B	
	JSR \$E8Ø3	Séquence "d'échappement"
	LDB # \$6D	
	JSR \$E8Ø3	2 ^e appel

On pourra alors superposer une image vidéo issue d'une source extérieure (télévision, magnétoscope, caméra) avec une partie d'écran, la couleur noire devenant "transparente".

Enfin, la valeur &H6C (au lieu de &H6D) permettra de supprimer l'incrustation.

II. Le clavier

Le clavier est organisé de manière matricielle ; le décodage des touches est effectué grâce au PIA système 6821.

Le port B (programmé en sortie) situé à l'adresse \$E7C9 fait passer l'une après l'autre chaque ligne à Ø ; si une touche de la ligne est enfoncée, la valeur lue sur le port A (programmé en entrée) situé en \$E7C8 est différente de &HFF, ce qui permet le décodage de la touche.

Nous verrons dans la quatrième partie de ce livre une routine permettant de décoder simultanément plusieurs touches appuyées en même temps, ce qui n'est pas possible avec les routines du moniteur.

1. Routine de lecture rapide: KTST\$

Cette routine, située en \$E8Ø9, teste si une touche est enfoncée, sans décodage de celle-ci ; le bit C du registre CC du 68Ø9 est alors positionné à 1.

Lors de l'exécution d'un programme BASIC, cette routine est appelée à chaque nouvelle instruction exécutée (voir 2^e partie) ; si elle retourne C=1, la touche est décodée par GETC\$ et le programme arrêté s'il s'agit de STOP ou CNT/C.

2. Décodage du clavier: routine GETC\$

Cette routine, située en \$E806, retourne dans l'accumulateur B du 6809 le code ASCII de la touche enfoncée, génère un "bip" sonore et gère la répétition des touches.

Elle est utilisée par les instructions BASIC de lecture du clavier, c'est-à-dire INKEY\$ et INPUT.

REMARQUE: la fonction INKEY\$ du BASIC TO7 retourne le dernier caractère frappé au clavier lors de l'exécution des instructions précédentes, même s'il ne l'est plus au moment de la lecture; on devra donc dans certains cas appeler un INKEY\$ "vide" pour effacer le registre \$65B1 mémorisant ce caractère (voir 2^e partie).

III. Affichages graphiques

Les deux routines PLOT\$ (située en \$E80F) et DRAW\$ (située en \$E80C) permettent respectivement d'afficher un point ou un caractère et de tracer une droite.

Elles sont utilisées par toutes les instructions graphiques du BASIC, c'est-à-dire PSET, LINE, BOX et BOXF.

1. Routine PLOT\$

Si le registre CHDRAW situé en \$6041 est mis à 0, la routine affiche le point de coordonnées contenues dans les registres X (colonne, de 0 à 319) et Y (ligne, de 0 à 199) du 6809.

La couleur du point doit être placée dans le registre FORME situé en \$6038; sa valeur doit être comprise entre - 8 et + 7 pour le TO7 et - 8 à + 15 pour le TO7-70.

Une valeur positive correspondra à un point de "forme" (bit correspondant mis à 1 dans la mémoire de forme: voir 2^e partie) et une valeur négative à un point de "fond" (bit mis à 0).

Les codes des couleurs "forme" sont les suivants :

noir	Ø
rouge	1
vert	2
jaune	3
bleu	4
magenta	5
cyan	6
blanc	7

Pour le TO7-7Ø, il existe huit codes supplémentaires :

gris	8
rose	: 9
vert clair	: 10
sable	: 11
bleu clair	: 12
parme	: 13
bleu ciel	: 14
orange	: 15

Dans les deux cas, un code de couleur "fond" sera obtenu en prenant l'inverse du code "forme" diminué de 1 : — 1 correspond au noir, — 2 au rouge, etc...

Si le registre CHDRAW contient un nombre C positif, le caractère de code ASCII C sera affiché en (X,Y), avec X compris entre 1 et 4Ø et Y entre Ø et 24 ; les couleurs de fond et de forme seront alors celles du registre COLOUR situé en \$6Ø3B.

On peut aussi écrire directement un caractère par un appel de la routine CHPL\$ située en \$E833.

2. Routine DRAW\$

Les conventions sont les mêmes que celles de la routine PLOT\$; la routine trace un segment de droite entre le dernier point allumé (registres PLOTX et PLOTY situés en \$6Ø3D et 6Ø3F) et le point de coordonnées (X,Y).

Si CHDRAW est différent de Ø, le segment tracé le sera sous forme de caractères.

REMARQUE: Pour le TO7-7Ø, le bit 4 du registre STATUS situé en \$6Ø19 doit être mis à Ø; dans le cas contraire, la mémoire "couleur" ne sera pas modifiée.

IV. Autres routines-registres du moniteur

On trouvera en annexe la liste complète des routines du moniteur, permettant d'effectuer toutes les opérations fondamentales: lecture du cryaon optique, génération de musique, entrée-sortie sur cassette, etc...

Toutes ces opérations seront réalisées beaucoup plus commodément à partir du BASIC; le détail des procédures à suivre pour un fonctionnement correct de ces routines ne sera donc pas décrit ici.

On trouvera aussi en annexe les adresses et le rôle des principaux registres du moniteur, tous situés à partir de l'adresse \$6ØØØ (page Ø du moniteur).

Tous ces registres sont bien sûr accessibles par le BASIC (instruction POKE), ce qui permet diverses interventions.

Sur le TO7-7Ø, on pourra par exemple redéfinir entièrement les touches du clavier; on placera pour cela dans le registre PTCLAV situé en \$6ØCD l'adresse d'une table (créée en BASIC sous la forme d'une chaîne de caractères) contenant les nouveaux codes ASCII des touches.

On pourra de la même manière redéfinir les caractères affichés sur l'écran; l'adresse du nouveau générateur de caractères sera placée pour cela dans le registre PTGENE situé en \$6ØCF.

3

Le 6809

Les deux ordinateurs TO7 de THOMSON sont conçus autour du très puissant micro-processeur 6809 de MOTOROLA.

Il s'agit en effet d'un micro-processeur 8 bits (bus de données à 8 lignes) capable de traiter des valeurs sur 16 bits (micro-processeur "pseudo 16 bits").

Il bénéficie de tous les progrès récents en matière de micro-processeurs : richesse des modes d'adressage, banalisation des registres internes, structure quasi orthogonale (la plupart des instructions fonctionnent avec tous les modes d'adressage), multiplication, etc...

1. Système binaire-codage hexadécimal

On sait que les ordinateurs ne peuvent utiliser que les valeurs 0 ou 1, appelées "bits".

Un ensemble de 8 bits constitue un "octet", qui est l'information de base manipulable par un micro-processeur 8 bits.

Un octet permet de coder $2^8 = 256$ valeurs différentes, qui pourront représenter une valeur numérique sur 8 bits, un caractère alphanumérique codé en ASCII (voir annexes), un code d'instruction, une partie d'adresse (toujours codées sur 16 bits), etc...

Exemples :

$$10110110 = 1 \times 2^7 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^1 = 182$$

La méthode la plus efficace pour noter le contenu d'un octet consiste à l'écrire en hexadécimal (base 16 et non 10 comme habituellement); la valeur sera alors représentée par deux chiffres hexadécimaux, codant chacun 4 bits.

Le codage est le suivant :

<i>Chiffre hexadécimal</i>	<i>Valeur binaire</i>	<i>Valeur décimale</i>
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Exemple :

$$10110110 = \&HB6$$

$$\&HE5 = 11100101$$

Arithmétique binaire :

Pour pouvoir représenter une valeur négative, on convient toujours que le bit de gauche, dit de "poids fort" car correspondant à 2^7 , représente le signe de la valeur.

Une valeur négative est alors représentée en "complément à 2" : la valeur absolue est complémentée bit à bit et on ajoute 1 (on peut aussi en hexadécimal retrancher la valeur absolue de &H100, ce qui revient au même).

Exemple :

$$58 = \&H3A \Rightarrow -58 = \&HC6$$

(voir annexe)

Dans le cas d'une valeur codée sur 2 octets, c'est le bit de poids fort du 1^{er} octet qui représente le signe.

Exemples :

$$14621 = \&H391D \Rightarrow -14621 = \&HC6E3$$

Une soustraction sur 8 bits est toujours effectuée en prenant le complément à 2 du 2^e opérande, que l'on ajoute au 1^{er}.

Une valeur positive et une valeur négative s'ajoutent "normalement".

Exemple : $\&H5C + \&HC6 = \&H22$, soit $92 - 58 = 34$

$$\begin{array}{rcl} \text{en effet :} & 01011100 & \longleftrightarrow \&H5C \\ & + 11000110 & \longleftrightarrow \&HC6 \\ \hline & 00100010 & \longleftrightarrow \&H22 \end{array}$$

$$\&H1B + \&HC6 = \&HE1, \text{ soit } 27 - 58 = -31$$

Exemple : $\&H1B + \&HC6 = \&HE1$, soit $27 - 58 = -31$

REMARQUE : le micro-processeur positionne certains indicateurs (retenue C et débordement V en particulier) à chaque opération effectuée.

On emploiera donc des branchements "signés" pour s'assurer de la validité des résultats obtenus en complément à 2.

On effectuera par contre des branchements "non signés" si on ne s'intéresse qu'aux valeurs absolues des résultats.

2. Les registres d'index X et Y

Ce sont des registres 16 bits permettant essentiellement de pointer des données en adressage indexé.

On peut aussi les utiliser pour effectuer des opérations arithmétiques.

3. Les pointeurs de pile U et S

Lorsqu'un programme appelle un sous-programme, le micro-processeur doit conserver quelque part en RAM l'adresse où il faudra revenir après l'exécution du sous-programme; il en est de même en cas d'interruption, où certains registres doivent aussi être conservés.

Ceci est toujours réalisé au moyen d'une *pile système*: seule la dernière valeur empilée, pointée par le registre S, est accessible (fonctionnement du type "last in, first out").

Le fonctionnement de la *pile utilisateur* U est exactement le même.

En cas d'empilement d'une nouvelle valeur ou adresse, le pointeur est *d'abord décrémenté*, puis il y a empilement.

Le dépilement s'effectue en sens inverse: la valeur située au sommet de la pile est dépilée dans le registre spécifié (PC dans le cas de RTS), puis le pointeur est incrémenté.

S et U pointent donc toujours sur *la valeur située au sommet* de la pile.

REMARQUE 1: dans le cas où on empile un registre 16 bits, c'est d'abord l'octet de poids faible qui est empilé, puis celui de poids fort; les deux octets sont donc placés en mémoire dans l'ordre logique, c'est-à-dire poids fort en α et faible en $\alpha + 1$, α étant contenu dans S (ou U).

REMARQUE 2: les registres U et S ont exactement les mêmes possibilités d'adressage indexé que X et Y.

4. Le registre de page directe DP

Ce registre 8 bits permet de placer la "page 0" n'importe où dans la mémoire; on peut ainsi faire de l'adressage "direct", toujours plus rapide et plus concis que l'adressage "absolu" normal.

5. Le registre de codes condition CC

Il s'agit d'un registre d'état de 8 bits, modifié à chaque instruction exécutée par le micro-processeur.

7	6	5	4	3	2	1	0
E	F	H	I	N	Z	V	C

Les bits 0, 1, 2, 3 (et 5) correspondent aux indicateurs arithmétiques, les plus souvent employés.

a) bit C ("carry": retenue)

Ce bit est positionné par toutes les opérations arithmétiques; il prend la valeur de la retenue dans le cas d'une addition (ADC, ADD), et celle du complément de la retenue dans le cas d'une soustraction (CMP, SBC, SUB).

Exemples : $\&H5C + \&HC6 \rightarrow C = 1$

$\&H5C - \&H3A \rightarrow C = 0$ (car $5C - 3A = 5C + C6 \rightarrow$ retenue = 1)

b) bit V ("Overflow": débordement en complément à 2)

Ce bit est mis à 1 lorsqu'une opération donne lieu à un débordement, c'est-à-dire lorsque la retenue des bits 6 (plus fort poids des valeurs signées) n'est pas la même que celle des bits 7 (bits de signe).

Exemples :

$\&H5C + \&HC6 = \&H22 \rightarrow V = 0$ car les deux retenues sont égales à 1 :
pas de problème

$\&H8C + \&HC6 \rightarrow V = 1$

débordement; le résultat ($\&H22$) est faux (en effet, $-116 - 58 = -174$, qui n'est pas représentable en complément à 2 sur 8 bits).

c) bit Z (zéro)

Il est mis à 1 si le résultat d'une opération (ou d'un chargement, ou d'un stockage, etc...) est nul :

Exemple :

CLRA ("clear A" : $0 \rightarrow A$) $\rightarrow Z = 1$

d) bit N ("Négative": résultat négatif)

Il représente le bit de poids fort du résultat de la dernière opération.

Il est donc mis à 1 dans le cas d'un résultat négatif, c'est-à-dire supérieur ou égal à &H80 en valeur absolue.

e) bit H ("Half carry": demi retenue)

Il permet au 6809 d'effectuer des opérations directement en décimal ; il est en effet positionné à 1 dès que le nombre présent sur les 4 bits de poids faible est supérieur à 9.

f) bits I et F ("Interrupt mask" et "Fast interrupt mask": masques d'interruption)

Lorsqu'ils sont mis à 1 (instruction ORCC), les interruptions normales IRQ et les interruptions rapides FIRQ (sauvegarde dans la pile des seuls PC et CC) sont masquées, c'est-à-dire ignorées.

g) bit E ("Entire state": pile complète)

En cas d'interruption, et lorsqu'il est à 1, tous les registres du 6809 ont été sauvegardés dans la pile (interruption IRQ).

6. Le registre compteur de programme PC

- Il s'agit du compteur ordinal, qui pointe en permanence sur *la prochaine instruction à exécuter*.
- Ce registre est utilisable comme index, ce qui permet par exemple d'écrire des programmes entièrement translatables.

III. Les modes d'adressage

Le 6809 possède six principaux modes d'adressage, ce qui lui confère une très grande puissance logicielle.

Dans tous les cas, les octets spécifiant l'adresse suivent le code opération, lui même codé sur 1 ou 2 octets.

1. Adressage inhérent ou implicite

Il n'y a alors pas de partie adresse puisque le code d'instruction se suffit à lui-même.

Exemple :

INCA : $A + 1 \rightarrow A$

CLRB : $\emptyset \rightarrow B$

Dans le cas des instructions d'échange et de transfert de registres (EXG et TFR), ou d'accès aux piles (PSH, PUL), l'octet spécifiant le code opération doit être toutefois suivi d'un "post-octet" indiquant les registres concernés (voir plus loin).

2. Adressage immédiat

La valeur à utiliser immédiatement suit le code opération ; selon ce dernier, elle occupera 8 bits (LDA, CMPB, etc...) ou 16 bits (LDX, CMPY, ADDD, etc...).

La notation normalisée d'une valeur immédiate est le dièse (#), en principe suivi d'un dollar (\$) signalant une valeur hexadécimale (notation "assembleur").

Exemple : Notation assembleur: LDX #\$830E

Valeurs en mémoire: 8E/83/0E

3. Adressage direct

Les 8 bits de poids fort de l'adresse de la valeur à traiter sont contenus dans le registre de page directe DP ; l'adresse est donc codée sur 1 seul octet représentant les 8 bits de poids faible.

Ce mode d'adressage est toujours plus rapide que l'adressage étendu.

Exemple : Notation assembleur: STD \$1C

Valeurs en mémoire: DD/1C

Si DP contient &H61, l'accumulateur D sera envoyé en \$611C (A) et \$611D (B).

REMARQUE: Le registre DP ne peut être chargé que par une instruction EXG ou TFR (ou PUL).

4. Adressage étendu

C'est l'adressage le plus classique ; l'adresse est codée sur 2 octets spécifiant l'emplacement de la mémoire où se situe la valeur à traiter.

Exemple : Notation assembleur : DEC \$60FE
Valeurs en mémoire : 7A/60FE

La valeur située en \$60FE est diminuée de 1.

5. Adressage relatif

Il est utilisé uniquement dans les opérations de branchements relatifs (BRA, LBEQ, BSR, etc...), réalisées le plus souvent après un test (BIT, CMP, TST).

L'adresse est ici égale à un déplacement positif ou négatif à ajouter au contenu du compteur de programme PC pour obtenir l'adresse effective du branchement.

Un déplacement codé sur 1 octet permettra d'avancer de 127(\$7F) octets ou de reculer de 128(\$80) octets au maximum ; il s'agit alors d'un branchement relatif court (BEQ, BRA, etc...).

Au-delà, on utilisera un branchement relatif long (mnémonique précédé de "L": LBRA, LBSR, etc...); le déplacement est alors codé sur 2 octets.

REMARQUE: lors de l'exécution de l'instruction de branchement, le PC pointe sur le début de l'instruction suivante ; c'est donc à cette adresse qu'il faut ajouter le déplacement.

Celui-ci sera bien sûr codé en mode complément à 2 pour une valeur négative : voir annexes.

Exemple : Soit les instructions suivantes, implantées à partir de \$8000.

8000	39	RTS	
8001	96 58	LDA	\$58
8003	27 05	BEQ	+5
8005	2B F9	BMI	-7
8007	BD 35EB	JSR	\$35EB
800A		...SUITE...	

Si \$58 (adressage direct) contient 0, il y aura branchement en \$8005 + 5 = \$800A = SUITE.

Si \$58 est négatif (bit 7 = 1), il y aura branchement en \$8007 - 7 = \$8000, c'est-à-dire au RTS.

Si \$58 contient une valeur comprise entre 1 et &H7F, il y aura exécution du sous-programme \$35EB (adressage étendu), puis passage à SUITE.

6. Adressages indexés

Il s'agit du mode d'adressage le plus puissant du 6809.

Le principe est que dans la partie adresse de l'instruction est toujours spécifié un index (appelé "base") et un déplacement.

L'adresse effective est alors égale au contenu de l'index augmenté du déplacement (éventuellement négatif).

Exemples : LDA 4,X: si X contient \$81E9, c'est la *valeur contenue en \$81ED*, soit \$3F par exemple, qui sera chargée dans l'accumulateur A.

LEAU - 5,X: c'est *l'adresse \$81E4* qui sera chargée dans U.

Dans le cas du 6809, le déplacement peut être constant (nul, ou codé sur 5, 8 ou 16 bits) ou variable (contenu d'un accumulateur A, B ou D); les index peuvent être indifféremment X, Y, U, S et PC, l'emploi de ce dernier autorisant par exemple de faire de l'adressage relatif indirect par rapport au PC (permettant d'écrire des programmes entièrement translatables)...

Il est possible aussi de faire des auto-incrémentations ou décrémentations de l'index de 1 ou 2 (le déplacement doit alors être nul) et de l'adressage indirect: l'adresse effective est alors l'adresse contenue à l'adresse obtenue par index + déplacement (écrit entre crochets).

Dans tous les cas, l'option choisie est déterminée par le "post-octet" suivant le code opération, lui-même suivi éventuellement de 1 ou 2 autres octets spécifiant le déplacement.

On trouvera en annexe le tableau complet de tous les cas possibles, permettant de déterminer le post octet.

Exemples :

LDA ,S ↔ A6/E4	: Valeur située au sommet de la pile → A
LDA -2,X ↔ A6/1E	: Déplacement -2 codé sur 5 bits
LDA 20,U ↔ A6/C8 14	: Déplacement +20 codé sur 8 bits
LDA B,Y ↔ A6/A5	: Déplacement contenu dans B

LDA ,X+ \leftrightarrow A6/80 : Déplacement nul; post-incrémentation de 1
 LDA ,--Y \leftrightarrow A6/A3 : Pré-décrémentation de 2; déplacement nul
 LDA [\$8010] \leftrightarrow A6/9F 8010 : Contenu de $\alpha \rightarrow \alpha A, \8010 et \$8011 contenant
 LDA [2,S] \leftrightarrow A6/F8 02 : Valeur située à l'avant dernière adresse placée dans la pile S A

IV. Le jeu d'instructions

Il se compose de 59 instructions fondamentales (se ramenant en fait à 56), permettant en fonction des registres utilisés et du mode d'adressage plus de 1400 combinaisons différentes !

On trouvera en annexe les tableaux décrivant toutes les instructions du 6809: mnémonique, code hexadécimal, nombre de cycles nécessaire pour l'exécution (1 cycle = 1 μ s. pour le 6809 des TO7), nombre total d'octets de l'instruction, action de l'instruction, bits du registre CC affectés.

Nous ne donnerons donc ici que les significations des mnémoniques (anglais bien sûr) et quelques compléments sur certaines instructions.

1. Les instructions

<i>Instruction</i>	<i>Description</i>
ABX	Addition de B (8 bits non signés) à X \rightarrow X
ADC	Addition avec retenue C
ADD	Addition 8 ou 16 bits
AND	ET logique (bit à bit)
ASL	Décalage arithmétique à gauche d'un bit
ASR	Décalage arithmétique à droite (bit 7 conservé)
BIT	Test de bit: effectue un ET, sans modifier l'accu.
CLR	Mise à 0 (clear)
CMP	Comparaison: effectue une soustraction, sans modifier le registre
COM	Complément bit à bit: 1 \rightarrow FE, etc...
CWAI	Attente d'interruption; effectue un ET avec le CC
DAA	Ajustement décimal de A (en 2 chiffres DCB)
DEC	Décrémentation de 1; 1 \rightarrow V si valeur initiale = &H80

<i>Instruction</i>	<i>Description</i>
EOR	OU exclusif (bit à bit: 1, 1 → 0)
EXG	Échange de registres (de même taille)
INC	Incrémentation de 1; 1 → V si valeur initiale = &H7F
JMP	Saut inconditionnel (jump)
JSR	Saut à un sous-programme; PC → pile S
LDx	Chargement du registre x (load)
LEA	Chargement d'une adresse effective (et non du contenu comme par LD) en adressage indexé
LSL	Décalage logique à gauche de 1 bit (logical shift left)
LSR	Décalage logique à droite (right) de 1 bit
MUL	Multiplication non signée de A par B → D
NEG	Complément à 2: 1 → FF = -1, etc...
NOP	Pas d'opération
OR	OU logique (bit à bit)
PSH	Empilement de registres (push); S décroît
PUL	Dépilement de registres (pull); S augmente
ROL	Rotation à gauche (avec C) de 1 bit
ROR	Rotation à droite (avec C) de 1 bit
RTI	Retour d'interruption; restauration des registres
RTS	Retour de sous-programme (restauration du PC)
SBC	Soustraction avec retenue C
SEX	Extension du signe de B à l'accumulateur D
STx	Stockage du registre x en mémoire
SUB	Soustraction
SWI	Interruption logicielle (sauvegarde des registres)
SYNC	Synchronisation avec un événement extérieur
TFR	Transfert de registres (de même taille)
TST	Test par rapport à 0 sur des valeurs non signées

REMARQUE: Les instructions ASL et LSL sont en fait identiques.

2. Les branchements relatifs

<i>Branchement</i>	<i>Description</i>
BCC, LBCC	Branchement si C = 0 (carry clear)
BCS, LBCS	Branchement si C = 1 (carry set)
BEQ, LBEQ	Branchement si égal, c'est-à-dire Z = 1 (equal)
BGE, LBGE	Branchement si supérieur ou égal (<i>signé</i> : branche si N = V, c'est-à-dire si résultat valide et positif ou nul)

<i>Branchement</i>	<i>Description</i>
BGT,LBGT	Branchement si supérieur (<i>signé</i> : branche si $N = V$ et $Z = \emptyset$)
BHI,LBHI	Branchement si supérieur, c'est-à-dire $C = \emptyset$ et $Z = \emptyset$ (higher)
BHS,LBHS	Branchement si supérieur ou égal, c'est-à-dire $C = \emptyset$ (higher or same)
BLE,LBLLE	Branchement si inférieur ou égal (<i>signé</i> : branche si $Z = 1$ ou $N \neq V$, c'est-à-dire si résultat valide et négatif ou null)
BLO,LBLO	Branchement si inférieur, c'est-à-dire $C = 1$ (lower)
BLS,LBLS	Branchement si inférieur ou égal, c'est-à-dire C ou $Z = 1$ (lower or same)
BLT,LBLT	Branchement si inférieur (<i>signé</i> : branche si $N \neq V$)
BMI,LBMI	Branchement si négatif, c'est-à-dire $N = 1$ (minus)
BNE,LBNE	Branchement si différent, c'est-à-dire $Z = \emptyset$ (not equal)
BPL,LBPL	Branchement si positif, c'est-à-dire $N = 0$, ou encore valeur $\leq \&H7F$
BRA,LBRA	Branchement inconditionnel (branch always)
BRN,LBRN	Ne branche jamais (branch never); créé par opposition à BRA
BSR,LBSR	Branchement à un sous programme (subroutine)
BVC,LBVC	Branchement si pas de débordement en mode complément à 2, c'est-à-dire $V = \emptyset$
BVS,LBVS	Branchement si débordement, c'est-à-dire $V = 1$ (V set)

REMARQUE: les branchements BCC et BHS (et LBCC et LBHS) sont en fait identiques, ainsi que BCS et BLO (et LBCS et LBLO).

3. Compléments

a) Instructions EXG et TFR

Elles permettent respectivement l'échange et le transfert de registre à registre.

Le code opération (1E et 1F) est suivi d'un post-octet spécifiant les registres concernés; le 1^{er} chiffre hexadécimal (bits de fort poids) représente le 1^{er} registre R1, le 2^e (faible poids) le registre R2 de destination (pour TFR).

Le codage est le suivant :

Code (hexa)	Registre
0	D
1	X
2	Y
3	U
4	S
5	PC
8	A
9	B
A	CC
B	DP

Exemples :

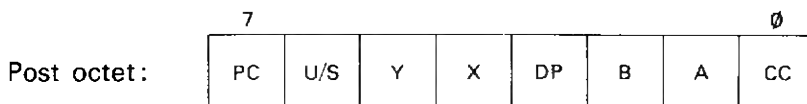
TFR S,X \leftrightarrow 1F/41 : S envoyé dans X

EXG U,D \leftrightarrow 1E/30 : U et D permutés

b) Instructions PSH et PUL

Elles permettent d'empiler ou de dépiler des registres.

Le code opération (&H34 à 37) est suivi d'un post octet spécifiant les registres concernés ; chaque bit correspond à un registre, qui sera empilé ou dépilé s'il a la valeur 1.



Exemple :

PSHS Y,X,A \leftrightarrow 34/32

Les registres sont toujours empilés dans l'ordre PC,U/S,...,CC ; le dépilement s'effectue en sens inverse.

S ou U ne peuvent être empilés dans leur propre pile ; c'est donc U qui est empilé par PSHS, et S par PSHU.

c) Instruction SWI

Elle permet de réaliser des interruptions logicielles ; elle est donc utilisée le plus souvent pour effectuer des arrêts sur adresse pour la mise au point de programmes.

L'instruction SWI provoque l'arrêt du programme en cours ; tous les registres (sauf S) sont sauvegardés dans la pile système S et il y a branchement à l'adresse contenue en \$FFFA et FFFB.

Les instructions SWI2 et SWI3 fonctionnent de manière identique (ce sont les interruptions les moins prioritaires) ; les vecteurs d'interruption sont situés respectivement en \$FFF4 et \$FFF2.

Dans le cas des deux TO7, et pour l'instruction SWI, l'adresse contenue en \$FFFA est celle d'une instruction JMP [\$602F] ; il suffit donc de placer dans le registre SWI1 du moniteur, situé en \$602F, l'adresse du sous-programme de traitement de l'interruption (qui écrira par exemple le contenu de la pile pour visualiser les registres) ; ce sous-programme se terminera par RTI.

4

Le langage machine

Le moyen le plus commode pour programmer en langage machine consiste bien sûr à utiliser un assembleur, permettant entre autres l'emploi des mnémoniques pour les instructions, l'écriture d'adresses symboliques, etc...

Ceci est exclu dans notre cas, puisque nous désirons utiliser le BASIC et que celui-ci ne peut coexister avec l'assembleur dans le cas des T07.

Nous allons voir qu'il est cependant relativement simple de créer à partir du BASIC des programmes écrits en langage machine.

Le problème inverse se pose pour étudier un programme tel que l'interpréteur ou le moniteur; nous présentons donc un programme BASIC permettant de "désassembler" du langage machine.

I. Implantation d'un programme

1. Méthode générale

Nous donnerons dans la troisième partie de cet ouvrage un programme permettant d'implanter n'importe où en mémoire en nombre quelconque de routines, et cela de la manière la plus souple et la plus "lisible" possible (un nom pour chaque routine, adresses écrites en un seul mot, etc...).

Pour de petites applications, il est parfaitement suffisant d'employer les quatre instructions suivantes :

```
10 CLEAR,&H7F00      'Mémoire Protégée à Partir de $7F01
20 FOR I=&H7F01 TO &H7FFF
30 READ A$:IF A$<>"FIN" THEN POKE I,VAL("&H"+A$):NEXT
1000 DATA            'Liste des codes,terminée Par FIN
```

On écrira alors en 1000 les codes hexadécimaux correspondant au programme, octet par octet, en les séparant par des virgules.

On enregistrera toujours sur cassette ou disquette le programme ci-dessus *avant* de faire exécuter le programme en langage machine ; en effet, en cas d'erreur dans celui-ci, le résultat est dans la plupart des cas un blocage complet de la machine, obligeant à couper le courant, d'où bien sûr perte du programme.

REMARQUES : CLEAR, α protège la zone de mémoire située à partir de $\alpha + 1$, qui contiendra le programme en langage machine.

En effet, la fin de la RAM est utilisée par le BASIC (entre autres pour les chaînes de caractères et la pile), qui pourrait donc sans cela "écraser" le programme.

On notera aussi l'usage d'une variable chaîne (A\$) pour charger les codes hexadécimaux, ce qui permet de ne pas écrire dans le DATA le fastidieux &H à chaque valeur.

2. Exemple

Soit le programme élémentaire suivant, dessinant un triangle au centre de l'écran :

	LDX	# \$5000	Adresse centre écran
	LDD	# \$8000	1 point à gauche
BOUCLE	STD	,X	Écriture dans mem.écran
	LEAX	40,X	Ligne suivante
	ORCC	# \$01	1 → C
	RORA		Construction ligne suivante
	RORB		
	BHS	BOUCLE	Ligne pas encore pleine
	RTS		Triangle terminé

Pour traduire ce programme en hexadécimal, il suffit de lire les tableaux donnés en annexe.

Par exemple, le code de LDX en adressage immédiat est 83 ; l'octet correspondant à l'adresse indexée notée ,X est 84 ; le branchement relatif à BOUCLE est codé — 11, soit F5 en complément à 2, puisque le PC pointe alors sur RTS (instruction suivante).

D'où le DATA :

```
1000 DATA 8E,50,0,CC,80,0,ED,84,30,88,28,1A,1,46,56,24,
F5,39,FIN
```

Après avoir fait exécuter le programme BASIC de création (par RUN), un simple EXEC &H7F01 dessinera un triangle (de côté 16 points) au centre de l'écran.

REMARQUE : Pour corriger un programme en langage machine, il suffit bien sûr de modifier le DATA, et ensuite de *refaire exécuter le programme de création* !...

II. Exécution-passage de paramètres avec le BASIC

1. Instruction EXEC

Cette instruction permet le branchement vers un sous-programme écrit en langage machine, qui doit se terminer par RTS.

Il n'est pas prévu ici d'échange de paramètres avec le programme BASIC ; il est toutefois possible de charger dans le sous-programme certaines adresses de la RAM avec des valeurs que l'on récupèrera par la fonction PEEK.

2. Instructions USR-DEFUSR

La fonction USR permet d'appeler un maximum de 10 sous-programmes différents écrits en langage machine, dont on aura défini préalablement l'adresse par une instruction DEFUSR n (n, facultatif, doit être compris entre 0 et 9).

L'appel sera réalisé en écrivant :

$V = \text{USRn}(x)$

La valeur x de l'argument est alors chargée automatiquement dans l'accumulateur flottant FAC dont l'adresse (\$6155: voir 2^e partie) est placée dans le registre X ; le type de l'argument est rangé dans l'accumulateur A du 6809 selon les conventions suivantes :

- 2 pour une valeur entière, que l'on lira en 2,X et 3,X
- 4 pour une valeur réelle, commençant en 0,X
- 8 pour une valeur double précision, en 0,X encore
- 3 pour une chaîne de caractères ; c'est alors l'adresse du descripteur de la chaîne qui est placée dans X ; on lira donc en 0,X la longueur de la chaîne, et en 1,X et 2,X l'adresse du premier caractère de la chaîne (voir 2^e partie).

Pour retourner une valeur dans la variable V, il faut bien sûr dans le sous-programme repositionner A et X avant le RTS, et placer la valeur soit en 0,X (types 3,4 et 8), soit en 2,X et 3,X (type 2 : entier).

REMARQUE: On peut aussi transmettre l'adresse α d'une variable V du BASIC par: `USRn (VARPTR(V))`.

VARPTR étant de type entier, α est alors rangée automatiquement en 2,X et 3,X.

On pourra ainsi traiter dans le sous programme les valeurs de plusieurs arguments; il suffit en effet pour celà de connaître la représentation mémoire des variables BASIC, que nous verrons dans la 2^e partie (elles sont toujours placées les unes après les autres, selon l'ordre de rencontre par l'interpréteur lors de l'exécution).

III. Désassemblage

Alors qu'il est relativement très simple et rapide de traduire "à la main" un programme assembleur en hexadécimal, le travail inverse est beaucoup plus long et fastidieux.

Il est de plus nécessaire de décoder un grand nombre d'instructions avant de pouvoir comprendre le fonctionnement d'un interpréteur ou d'un moniteur !

Nous donnons donc ici un programme BASIC réalisant automatiquement ce désassemblage.

1. Le programme

A partir d'une suite de codes hexadécimaux, il fournit sur l'écran ou sur l'imprimante un listing assembleur normalisé.

Le listing est le suivant :

```
57000 'Initialisation des tableaux...
57010 DIM COD$(255),COD2$(255),N(255)
57040 READ C$:FOR N=1 TO 88
57050 READ A$:IF LEN(A$)<3 THEN ADR=VAL("&H"+A$):COD$(ADR)=C$:READ N(ADR):GOTO57050
57060 C$=A$:NEXT N
57110 FOR N=1 TO 7
57120 READ A$:IF LEN(A$)<3 THEN ADR=VAL("&H"+A$):COD2$(ADR)=C$:GOTO57120
57130 C$=A$:NEXT N
57210 RESTORE 59600:FOR N=1 TO 15
57220 READ C$,A$:ADR=VAL("&H"+A$)
57230 COD$(ADR)=C$:N(ADR)=5
```

```

57240 COD2$(ADR)="L"+C$:NEXT N
57250 COD$(22)="LBRA":N(22)=6
57260 COD$(23)="LSR":N(23)=6
57270 COD$(32)="BRA":N(32)=5
57280 COD$(141)="BSR":N(141)=5
57390 '
57395 'Desassemblage...
57400 INPUT " (-)Adresse de depart(Hexa.)":A$:CLOSE 1
57410 IF LEFT$(A$,1)<>"-" THEN OPEN "0",1,"SCRN":ADRF=
&HFFFF:LMAX=23 ELSE OPEN "0",1,"LPT":A$=MID$(A$,2):LM
AX=99:INPUT " Adresse de fin(Hexa.)":C$:ADRF=VAL("
&H"+LEFT$(C$,4))
57420 CLS:ER=0:ADR=VAL("&H"+LEFT$(A$,4))
57500 FOR L=0 TO LMAX
57505 IF ADR>ADRF GOT057400
57510 IND=0:COD2=0:PRINT#1,HEX$(ADR);
57520 COD=PEEK(ADR):PRINT#1,TAB(8);HEX$(COD);
57530 IF ER=1 THEN N=1:GOT058550
57535 '
57540 REM.....Traitement d'un code...
57550 IFCOD=16 OR COD=17 THEN GOSUB58700 ELSE C$=COD$(C
OD)
57560 IFC$="" THEN ER=1:N=1:PRINT#1,TAB(13);"???Code ine
xistant":GOT058550
57600 REM...Traitement de l'adresse...
57610 N=N(COD):PRINT#1,TAB(13);X=PEEK(ADR+1)
57620 IF N>4 GOT057800 'branchement
57630 IF N=0 GOT057900 'Indexation
57640 A$=LEFT$(C$,3):IF A$="PSH" OR A$="PUL" GOT058340
57650 IF A$="EXG" OR A$="TFR" GOT058400
57660 IF N<0 THEN IND=-1:N=-N
57700 REM..Ecriture de l'instruction...
57710 GOSUB58800:PRINT#1,TAB(21);C$:TAB(27);
57720 IF IND<0 THEN PRINT#1,"#";
57730 IF N>1 THEN PRINT#1,"$";
57740 GOSUB58800:GOT058550 '...Suite
57795 '
57800 REM...Branchement relatif....
57810 IF N=6 OR COD2=1 GOT057850 'Long
57820 N=2:IF X>127 THEN X=X-256
57830 X=ADR+2+X:GOT057870
57850 N=3:IF X>127 THEN X=X-256
57860 X=256*X+PEEK(ADR+2)+ADR+3
57870 A$="$"+HEX$(X):GOT058500
57895 '
57900 REM.....Adresse indexe.....
57910 IF X>127 GOT058000
57920 GOSUB58920:V=X AND &H10 'Bit 4

```

```

57930 X=X AND 15:IF V>0 THEN X=X-16
57940 A$=STR$(X)+", "+A$:N=2:GOTO58500
58000 IF(X AND &H1F)=&H1F THEN A$=CHR$(91)+"$"+HEX$(PEEK(ADR+2)):N=4:IND=3:GOTO58500
58010 GOSUB58900:IF(X AND &H10)=0 THEN IND=1 ELSE IND=2
58020 X=X AND 15:IF X>6 AND X<>11 GOTO58100
58030 N=2:ON X+1 GOTO58045,58050,58055,58060,58065,58070,58075
58040 A$="D, "+A$:GOTO58240
58045 A$=" ", "+A$+"+":GOTO58240
58050 A$=" ", "+A$+"+":GOTO58240
58055 A$=" ", "-"+A$:GOTO58240
58060 A$=" ", "--"+A$:GOTO58240
58065 A$=" ", "+A$:GOTO58240
58070 A$="B, "+A$:GOTO58240
58075 A$="A, "+A$:GOTO58240
58100 IF X<>8 AND X<>12 GOTO58200
58110 X=PEEK(ADR+2):IF X>127 THEN X=X-256
58120 A$=STR$(X)+", "+A$:N=3:GOTO58240
58200 IF X<>9 AND X<>13 GOTO58300
58210 N=4:X=PEEK(ADR+2):IF X>127 THEN X=X-256
58220 X=256*X+PEEK(ADR+3)
58230 A$=STR$(X)+", "+A$
58240 IF IND=2 THEN A$=CHR$(91)+A$+CHR$(93)
58250 GOTO58500
58300 ER=1:N=2:A$="???Index":GOTO58500
58335 '
58340 REM...PSHx ou PULx...
58345 A$="":V=X MOD 2:IF V=1 THEN A$=A$+",CC"
58350 X=X/2:V=X MOD 2:IF V=1 THEN A$=A$+",A"
58355 X=X/2:V=X MOD 2:IF V=1 THEN A$=A$+",B"
58360 X=X/2:V=X MOD 2:IF V=1 THEN A$=A$+",DP"
58365 X=X/2:V=X MOD 2:IF V=1 THEN A$=A$+",X"
58370 X=X/2:V=X MOD 2:IF V=1 THEN A$=A$+",Y"
58375 X=X/2:V=X MOD 2:IF V=0 GOTO58385
58380 IF RIGHT$(C$,1)="S" THEN A$=A$+",U" ELSE A$=A$+",S"
58385 X=X/2:IF X=1 THEN A$=A$+",PC"
58390 A$=MID$(A$,2):GOTO58500
58400 REM...TFR ou EXG...
58410 V=X AND 15:X=X/16:A$=""
58415 ON X+1 GOTO58425,58430,58435,58440,58445,58450,58455,58460,58465,58470
58420 A$="???Erreur":ER=1:GOTO58500
58425 A$=A$+"D":GOTO58480
58430 A$=A$+"X":GOTO58480

```

```

58435 A$=A$+"Y":GOTO58480
58440 A$=A$+"U":GOTO58480
58445 A$=A$+"S":GOTO58480
58450 A$=A$+"PC":GOTO58480
58455 A$=A$+"A":GOTO58480
58460 A$=A$+"B":GOTO58480
58465 A$=A$+"CC":GOTO58480
58470 A$=A$+"DP"
58480 IF IND=0 THEN IND=1:A$=A$+",":X=Y:GOTO58415
58495 '
58500 REM...Ecriture de l'instruction..
58510 GOSUB58800:PRINT#1,TAB(21);C$;
58515 IF LEFT$(A$,1)=" " THEN A$=MID$(A$,2)
58520 PRINT#1,TAB(27);:IF LEN(A$)>9 THEN PRINT#1:PRINT#
1,TAB(17);:L=L+1
58525 PRINT#1,A$;:IF IND<3 GOTO58550
58530 X=PEEK(ADR+3):IF X<16 THEN PRINT#1,"0";
58540 PRINT#1,HEX$(X);CHR$(93);
58550 REM....Instruction suivante.....
58560 ADR=ADR+N
58570 PRINT#1:NEXT L:GOTO57400      'Fin...
58695 '
58700 REM....Code op. sur 2 octets.....
58710 X=COD:ADR=ADR+1:COD=PEEK(ADR)
58720 PRINT#1,HEX$(COD);:C$=COD2$(COD)
58730 IF X=16 THEN COD2=1:RETURN
58740 IF C$<>"CMPD" AND C$<>"CMPY" AND C$<>"SWI2" THEN
C$="":RETURN
58750 IF C$="SWI2" THEN C$="SWI3":RETURN
58760 IF C$="CMPD" THEN C$="CMPU" ELSE C$="CMPS"
58770 RETURN
58795 '
58800 REM....Ecriture d'une adresse....
58810 FOR I=1 TO N-1
58820 V=PEEK(ADR+I):IF V<16 THEN PRINT#1,"0";
58830 PRINT#1,HEX$(V);
58840 NEXT I:RETURN
58895 '
58900 REM...Determination de l'index...
58910 IF(X AND12)=12 THEN A$="PC":RETURN
58920 V=X AND &H60:IF V=0 THEN A$="X":RETURN
58930 IF V=&H20 THEN A$="Y":RETURN
58940 IF V=&H40 THEN A$="U" ELSE A$="S"
58950 RETURN

```



```

58990 /
58995 REM Instructions 6809....
59000 DATA ABX,3A,1
59005 DATA ADCA,89,-2,99,2,A9,0,B9,3
59010 DATA ADCB,C9,-2,D9,2,E9,0,F9,3
59015 DATA ADDA,8B,-2,9B,2,AB,0,BB,3
59020 DATA ADCB,CB,-2,DB,2,EB,0,FB,3
59025 DATA ADDD,C3,-3,D3,2,E3,0,F3,3
59030 DATA ANDA,84,-2,94,2,A4,0,B4,3
59035 DATA ANDB,C4,-2,D4,2,E4,0,F4,3
59040 DATA ANDCC,1C,-2
59045 DATA ASRA,47,1,ASRB,57,1
59055 DATA ASR,7,2,67,0,77,3
59060 DATA BITA,85,-2,95,2,A5,0,B5,3
59065 DATA BITB,C5,-2,D5,2,E5,0,F5,3
59070 DATA CLRA,4F,1,CLRB,5F,1
59080 DATA CLR,F,2,6F,0,7F,3
59085 DATA CMPA,81,-2,91,2,A1,0,B1,3
59090 DATA CMPB,C1,-2,D1,2,E1,0,F1,3
59095 DATA CMPX,8C,-3,9C,2,AC,0,BC,3
59100 DATA COMA,43,1,COMB,53,1
59110 DATA COM,3,2,63,0,73,3
59115 DATA CMAL,3C,-2
59120 DATA DAA,19,1
59125 DATA DECA,4A,1,DECB,5A,1
59135 DATA DEC,A,2,6A,0,7A,3
59140 DATA EORA,88,-2,98,2,A8,0,B8,3
59145 DATA EORB,C8,-2,D8,2,E8,0,F8,3
59150 DATA EXG,1E,2
59155 DATA INCA,4C,1,INCB,5C,1
59165 DATA INC,C,2,6C,0,7C,3
59170 DATA JMP,E,2,6E,0,7E,3
59175 DATA JSR,9D,2,AD,0,BD,3
59180 DATA LDA,86,-2,96,2,A6,0,B6,3
59185 DATA LDB,C6,-2,D6,2,E6,0,F6,3
59190 DATA LDD,CC,-3,DC,2,EC,0,FC,3
59195 DATA LDU,CE,-3,DE,2,EE,0,FE,3
59200 DATA LDX,8E,-3,9E,2,AE,0,BE,3
59205 DATA LEAS,32,0,LEAU,33,0
59215 DATA LEAX,30,0,LEAY,31,0
59225 DATA LSLA,48,1,LSLB,58,1
59235 DATA LSL,8,2,68,0,78,3
59240 DATA LSRA,44,1,LSRB,54,1
59250 DATA LSR,4,2,64,0,74,3
59255 DATA MUL,3D,1
59260 DATA NEGA,40,1,NEGB,50,1
59270 DATA NEG,0,2,60,0,70,3
59275 DATA NOP,12,1

```

59280 DATA ORA, 8A, -2, 9A, 2, AA, 0, BA, 3
 59285 DATA ORB, CA, -2, DA, 2, EA, 0, FA, 3
 59290 DATA ORCC, 1A, -2
 59295 DATA PSHS, 34, 2, PSHU, 36, 2
 59305 DATA PULS, 35, 2, PULU, 37, 2
 59315 DATA ROLA, 49, 1, ROLB, 59, 1
 59325 DATA ROL, 9, 2, 69, 0, 79, 3
 59330 DATA RORA, 46, 1, RORB, 56, 1
 59340 DATA ROR, 6, 2, 66, 0, 76, 3
 59345 DATA RTI, 38, 1, RTS, 39, 1
 59355 DATA SBCA, 82, -2, 92, 2, A2, 0, B2, 3
 59295 DATA PSHS, 34, 2, PSHU, 36, 2
 59305 DATA PULS, 35, 2, PULU, 37, 2
 59315 DATA ROLA, 49, 1, ROLB, 59, 1
 59325 DATA ROL, 9, 2, 69, 0, 79, 3
 59330 DATA RORA, 46, 1, RORB, 56, 1
 59340 DATA ROR, 6, 2, 66, 0, 76, 3
 59345 DATA RTI, 38, 1, RTS, 39, 1
 59355 DATA SBCA, 82, -2, 92, 2, A2, 0, B2, 3
 59360 DATA SBCB, C2, -2, D2, 2, E2, 0, F2, 3
 59365 DATA SEX, 1D, 1
 59370 DATA STA, 97, 2, A7, 0, B7, 3
 59375 DATA STB, D7, 2, E7, 0, F7, 3
 59380 DATA STD, DD, 2, ED, 0, FD, 3
 59385 DATA STU, DF, 2, EF, 0, FF, 3
 59390 DATA STX, 9F, 2, AF, 0, BF, 3
 59395 DATA SUBA, 80, -2, 90, 2, A0, 0, B0, 3
 59400 DATA SUBB, C0, -2, D0, 2, E0, 0, F0, 3
 59405 DATA SUBD, 83, -3, 93, 2, A3, 0, B3, 3
 59410 DATA SWI, 3F, 1, SYNC, 13, 1
 59420 DATA TFR, 1F, 2
 59425 DATA TSTA, 4D, 1, TSTB, 5D, 1
 59435 DATA TST, D, 2, 6D, 0, 7D, 3
 59500 DATA CMPD, 83, 93, A3, B3
 59505 DATA CMPY, 8C, 9C, AC, BC
 59510 DATA LDS, CE, DE, EE, FE
 59515 DATA LDY, 8E, 9E, AE, BE
 59520 DATA STS, DF, EF, FF
 59525 DATA STY, 9F, AF, BF
 59530 DATA SWI2, 3F
 59600 DATA BHS, 24, BLO, 25, BEQ, 27
 59605 DATA BGE, 2C, BGT, 2E, BHI, 22
 59610 DATA BLE, 2F, BLS, 23, BLT, 2D
 59615 DATA BMI, 28, BNE, 26, BPL, 2A
 59620 DATA BRN, 21, BVC, 28, BVS, 29

REMARQUE: La longueur relativement impressionnante du listing, et en particulier celle des DATA, est bien sûr due à la richesse du jeu d'instructions et des modes d'adressage du 6809 !

Dans un premier temps, on pourra d'ailleurs ne pas écrire les 25 instructions situées de 58345 à 58480, en ajoutant seulement:

```
58400 A$=""xxx"
```

Les registres concernés par TFR, EXG,PSH et PUL ne seront alors pas décodés.

Commentaires sur le programme :

Le programme se compose en fait de deux parties distinctes.

Les lignes de 57000 à 57280 et celles situées à partir de 59000(DATA) servent à initialiser les tableaux COD\$ et COD2\$ contenant les mnémoniques de toutes les instructions, codées respectivement sur 1 et 2 octets; le tableau N est en même temps initialisé avec le nombre total d'octets de l'instruction: une valeur négative indique un adressage immédiat, la valeur 0 un adressage indexé (adresse codée sur 1,2 ou 3 octets, en fonction du post-octet) et une valeur supérieure ou égale à 5 un adressage relatif.

Ces tableaux sont ensuite utilisés par le programme de désassemblage proprement dit, constitué par les lignes de 57400 à 58950.

Pour chaque instruction du programme en langage machine, il y a d'abord lecture du code COD d'instruction, d'où détermination du mnémonique C\$ situé dans COD\$(COD) ou COD2\$(COD).

La partie adresse est ensuite traitée, en fonction de la valeur de N(COD).

Cas du TO7 sans extension mémoire :

Les deux parties du programme devront être dissociées si l'on ne dispose que des 8 K de la version de base.

On ne conservera donc pour le 1^{er} programme que les lignes de 57000 à 57280 et celles à partir de 59000, auxquelles on ajoutera les instructions suivantes :

```
57300 OPEN"O",2,"TABLEAUX"  
57310 FOR I=0 TO 255  
57320 PRINT#2,COD$(I),COD2$(I),N(I):NEXT  
57330 END
```

Ce programme enregistrera donc une fois pour toutes sur cassette les tableaux COD\$, COD2\$ et N (appuyer sur la touche "Enregistrement" du magnétophone avant de faire RUN).

Le programme de désassemblage proprement dit sera alors constitué des seules lignes 57000 et 57400 à 58950, auxquelles on ajoutera la lecture des tableaux :

```
57100 OPEN"1",2,"TABLEAUX"
57110 FOR I=0 TO 255
57120 INPUT#2,COD$(I),COD2$(I),N(I):NEXT
```

2. Mode d'emploi-résultats

Lorsque le programme demande l'adresse de départ, on tapera directement l'adresse hexadécimale du début du désassemblage ; on obtiendra alors sur l'écran 24 instructions décodées, et une demande de nouvelle adresse.

Si on désire conserver le listing assembleur sur imprimante, on fera simplement précéder l'adresse initiale du signe "--"; le programme demande alors l'adresse de fin, que l'on tapera en hexadécimal toujours.

La validité des adresses est bien sûr systématiquement contrôlée.

Si on reprend l'exemple du programme dessinant un triangle, implanté ici en \$BF01, on obtient :

BF01	8E	5000	LDX	#\$5000
BF04	CC	8000	LDD	#\$8000
BF07	ED	84	STD	,X
BF09	30	8828	LEAX	40,X
BF0C	1A	01	ORCC	#\$01
BF0E	46		RORA	
BF0F	56		RORB	
BF10	24	F5	BHS	\$BF07
BF12	39		RTS	

La première colonne contient les adresses hexadécimales du début de chaque instruction ; puis on a les octets correspondant à cette dernière (1 à 5 octets au maximum), et enfin l'instruction décodée, écrite en notation assembleur normalisée.

Pour les branchements relatifs, c'est l'adresse effective du branchement qui est donnée.

Pour l'instruction PULx, les registres sont dépilés dans l'ordre listé (exemple: \$65B PULS A,X,U); par contre, l'empilement (PSHx) s'effectue en fait dans l'ordre inverse.

Les registres transférés ou échangés sont listés en clair (*Exemple*: \$1599 TFR S,X), ainsi que tous les adressages indexés.

Enfin, rappelons que # désigne un adressage immédiat, \$ un nombre hexadécimal et les crochets [] un adressage (indexé ou non) indirect.

Deuxième partie

L'INTERPRÉTEUR BASIC MICROSOFT

Notre but n'est pas de donner ici une simple liste d'adresses de routines, ou de renseignements, sur l'interpréteur BASIC des micro-ordinateurs THOMSON T07 et T07-70.

Nous décrivons tout au contraire une *méthode* permettant de décoder n'importe quel interpréteur Microsoft, ceux-ci étant conçus toujours de la même (excellente !) manière.

Une fois compris et décrit le fonctionnement de l'interpréteur (chapitre I et II), nous étudions (chapitre III) le traitement de quelques instructions fondamentales du BASIC.

Nous signalons enfin que certaines routines détaillées ci-après demandent un effort de compréhension ; le lecteur se verra récompensé par toutes les applications qui découleront de cette étude.

1

Comprendre l'exécution d'un programme BASIC

Les instructions d'un programme sont décodées et exécutées par une routine qui se présente sous la forme d'une boucle, décrite à chaque nouvelle instruction.

Grâce à une table des adresses, l'interpréteur détermine, à partir du code de l'instruction BASIC à exécuter, l'adresse du sous programme correspondant, qui réalise le traitement; l'instruction suivante est alors prise en compte pour poursuivre l'exécution du programme.

Une fois décrit le codage d'un programme, puis trouvée la table des adresses et la boucle d'exécution, nous aurons tous les éléments nécessaires à la compréhension du fonctionnement de l'interpréteur.

I. Implantation et codage d'un programme

1 — Pour trouver l'adresse à partir de laquelle est implanté un programme, on peut par exemple utiliser le programme suivant :

```
10 REM Programme 1
20 AB=0 'AB Code &H41,42
30 FOR I=&H6000 TO &H7FFF 'RAM
40 IF PEEK(I)=&H41 THEN IF PEEK(I+1)=&H42 THEN PRINT HE
X$(I)
50 NEXT:END
```

Pour les TO7, on obtient les adresses \$660C et \$6619 (et aussi \$6680, situé dans la zone des variables: voir chapitre suivant); le programme commence donc une vingtaine d'octets auparavant, soit exactement en \$65F5.

2 — Le programme suivant permettra alors de visualiser en hexadécimal le codage d'un programme :

```
100 REM Programme 2
110 ADR0=&H65F5
120 CLS:FOR I=ADR0 TO ADR0+229 STEP 10
130 PRINT HEX$(I)+";":FOR J=I TO I+9
140 PRINT USING"% 2";HEX$(PEEK(J));:NEXT
150 PRINT:NEXT I:END
```

On obtient :

65F5	66	8	0	64	8C	20	20	50	72	6F	67	72
6601	61	6D	6D	65	20	32	0	66	18	0	6E	41
660D	44	52	30	D4	26	48	36	35	46	35	0	66
6619	37	0	78	9D	3A	81	20	49	D4	41	44	52
6625	30	20	BB	20	41	44	52	30	C7	20	35	39

3 — Connaissant le code ASCII (Annexes), on peut en déduire les éléments suivants :

— Chaque ligne possède un en-tête de 4 octets :

- Les deux premiers contiennent l'adresse de l'en-tête de la ligne suivante;
- Les deux suivants contiennent le numéro de la ligne.

- Chaque mot BASIC (REM, =, CLS, FOR, etc...) est codé sur un ou deux octets (le premier étant alors égal à &HFF), de valeur supérieure ou égale à &H80.
- Chaque ligne se termine par un octet contenant 0, qui précède l'en-tête de la ligne suivante.
- La fin du programme est marquée par trois zéros consécutifs (ici en \$6677); ils sont suivis de la zone où l'on trouve les variables.

II. La table et les codes des mots réservés du BASIC

1. Recherche de la table

Lors de la saisie des instructions d'un programme, l'interpréteur reconnaît et code les différents mots du BASIC, rangés dans une table par ordre de codes croissants.

Celle-ci contient les codes ASCII des caractères composant les mots; la dernière lettre est toutefois codée différemment pour marquer la fin de chaque mot.

La table sera donc trouvée à l'aide du programme 1 précédent, en recherchant dans la ROM BASIC (adresses de \$0 à \$3FFF) les caractères EN (codés &H45,4E) du mot END situé au début de la table (code &H80).

On obtient \$92,\$162 et \$1758; mais un examen de ces adresses (programme 2 précédent, en modifiant ADR0) montre que le END (suivi de FOR, de code &H81) est bien en \$92.

Le début de la table est donc en \$92, et la fin en \$269; \$26A contient un 0 marquant la fin de la table.

On constate de plus que le bit de plus fort poids du dernier caractère de chaque mot est mis à 1 dans la table.

Le programme suivant permet de lister les 125 mots du BASIC, avec leurs codes (le programme 2, où l'on écrit différentes instructions et fonctions à la ligne 20, permet de constater que le codage change après le 87^e mot, codé &HD5).

~

```

200 REM Programme 3
210 A$="":CODE=&H80
220 FOR I=&H92 TO &H269
230 X=PEEK(I):IF X<&H80 THEN A$=A$+CHR$(X):GOTO260
240 A$=A$+CHR$(X-&H80):IF CODE<=&HD5 THEN PRINT HEX$(CODE); ELSE PRINT"FF"+HEX$(CODE-86);
250 PRINT": "+A$, :CODE=CODE+1:A$=""
260 NEXT:END

```

Le résultat est donné en annexe; on constatera que les fonctions sont codées sur 2 octets, le premier étant égal à &HFF.

La table des mots est donc en fait composée de 2 tables:

- celle des instructions commence en \$92 et comporte 86 mots (&H56),
- celle des fonctions commence en \$1CF (trouvé toujours grâce au programme 1, en recherchant les caractères SG de SGN) et comporte 39 mots (&H27).

2. Utilisation par l'interpréteur

L'interpréteur utilise la table des mots en initialisant un des registres avec l'adresse du début de la table; ce registre r est ensuite incrémenté pour balayer la table, jusqu'à ce que soit trouvée la suite des caractères composant le mot à coder.

L'initialisation du registre peut être à priori faite soit par l'instruction *LDr # \$0092* ou *\$01CF* pour les fonctions), soit par une instruction *LDr adresse*, "adresse" étant l'adresse d'une mémoire contenant \$0092 (ou \$01CF).

Le programme suivant recherche la valeur &H0092 dans la ROM et dans la RAM (de \$6000 à \$65F4):

```

300 REM Programme 4
310 A=0:B=&H92
320 I0=0:I1=&H3FFF:GOSUB350
330 I0=&H6000:I1=&H65F4:GOSUB350
340 BEEP:END
345 '
350 FOR I=I0 TO I1
360 IF PEEK(I)=A THEN IF PEEK(I+1)=B THEN PRINT HEX$(I)
,HEX$(PEEK(I-1)),HEX$(A)+HEX$(B)
370 NEXT:RETURN

```

On trouve \$38AA et \$6202, précédés tous deux de &H56 (code de RORB, qui ne convient pas); pour la valeur &H1CF (obtenue avec 20 A=1: B=&HCF), on trouve \$38AF et \$6207, précédés tous deux de &H27 (code de BEQ, ce qui ne convient pas davantage).

Pour l'initialisation du début des tables, on doit donc chercher une instruction ayant \$38AA ou \$6202 comme partie adresse, puis \$38AF ou \$6207 (on verra que la page 0 du BASIC n'est pas située en \$62, et on n'a donc pas à chercher une adresse \$02 ou \$07).

Le programme 4 permet (en modifiant la ligne 310) de constater que l'on a nulle part une telle instruction.

On peut pourtant constater que la zone de \$6201 à \$620A contient les mêmes informations que la zone de \$38A9 à \$38B2; cette dernière zone est donc recopiée dans la RAM à l'initialisation du système (par une routine située en \$37E5), dans le but évident d'être utilisée plus tard !...

On doit donc chercher dans l'interpréteur les 2 instructions:

LD r # valeur

LEA r d,r (ou ADD # d si r=D)

avec d=\$6202 — valeur (ou \$6207 — valeur)

Le programme 4 légèrement modifié permet de lister toutes les instructions ayant une adresse de \$61F0 à \$620F par exemple; on en trouve 9, dont seulement 4 sont précédées d'un code d'instruction à adressage immédiat (LDU dans les quatre cas); il s'agit de \$2969, 2A1F, 2A60 et 37E9.

Un examen attentif, par le désassembleur de la première partie, des instructions situées autour de ces adresses montre que les instructions:

2A1E LDU # \$61F7,
2A5F LDU # \$61FC,

correspondent respectivement à l'initialisation des registres Y et B par les valeurs situées en \$6202 (adresse du début de la table) et \$6201 (nombre de mots), et par celles situées en \$6207 et \$6206.

Dans les deux cas, il y a en effet utilisation de la routine suivante, située en \$2A21:

2A21	CLR	\$45	Numéro du mot dans la table
	LEAU	10,U	\$6201 ou 6206
	LDB	,U	Nombre de mots dans la table

	BEQ	\$2A5F	
	LDY	1,U	Contenu de \$6202 ou 6207 → Y
	LDX	,S	Adresse début du mot du buffer
2A2E	LDA	,X+	Une lettre du mot à coder
	BSR	\$2A88	Minuscule → majuscule
	SUBA	,Y+	Une lettre de la table
	BEQ	\$2A2E	Lettre suivante
	CMPA	# \$80	Si égal, on a trouvé le mot
	BNE	\$2A76	Mot suivant
	etc...		

A titre indicatif, signalons que le codage d'une instruction est réalisé par une routine commençant en \$29A3, appelée elle-même en \$44C (ou en \$424 si on est en mode commande) ; cette routine code les mots du BASIC directement dans le buffer clavier contenant les caractères de l'instruction, situés à partir de \$6445 ; le codage est effectué lors de la frappe de la touche "ENTRÉE" terminant l'instruction.

Signalons aussi que la routine contenant l'instruction \$2968 est utilisée par LIST ; elle opère en sens inverse de \$29A3.

3. Application immédiate

Il suffit de modifier (POKE ou LOADM) le contenu des octets \$6201 à 6203, et \$6206 à 6208, pour pouvoir créer son propre vocabulaire BASIC, par exemple avec des mots français : voir troisième partie.

III. La table des adresses d'instructions

La table des adresses de traitement des instructions et fonctions contient forcément une suite d'adresses de routines situées dans le BASIC ; le 1^{er} octet doit donc être toujours inférieur ou égal à &H3F (ROM) ou, éventuellement, compris entre &H60 et &H65 (RAM avant le programme).

Le programme 5 suivant cherche donc dans la ROM une suite d'au moins 20 couples d'octets consécutifs dont le 1^{er} est inférieur ou égal à &H3F.

```

400 REM Programme 5
410 FOR I=0 TO &H3FE0
420 IF PEEK(I)>&H3F GOTO460
430 FOR J=I+2 TO I+248 STEP 2      '250 adresses maxi

```

```

440 IF PEEK(J)<&H40 THEN NEXT
450 IF J>I+40 THEN PRINT HEX$(I), HEX$(J-1): I=J+1    '>20
460 NEXT I

```

On obtient \$1C à \$6D et \$26B à \$2C8 ; or, si l'on observe (programme 2) la fin de cette dernière zone, on s'aperçoit en fait qu'elle continue jusqu'en \$2DE (soit 58 adresses), puisque jusque-là on ne trouve qu'en \$2C9 et \$2CB les adresses \$6233 et \$6236, adresses où l'on a JMP \$7F3 renvoyant à une adresse de la ROM.

Or, on pourra constater que seuls les 58 premiers mots du BASIC (de END à PLAY) sont directement exécutables.

Une ultime vérification consiste à relever par exemple les 1^{ère}, 9^e, 30^e et 36^e adresses de la table, soient \$53B, \$5F2, \$35EB et \$35E6, correspondant en principe respectivement à END, RUN, CLS et BEEP.

- EXEC &H53B inséré dans un programme provoque l'arrêt de celui-ci ;
- EXEC &H5F2 provoque le redémarrage du programme à partir de la première ligne ;
- EXEC &H35EB efface l'écran ;
- EXEC &H35E6 génère un "bip" sonore.

La table des adresses de traitement des instructions se situe donc bien en \$26B ; on trouvera ces adresses en annexe.

Nous verrons au chapitre suivant que la zone de \$20 à \$6D représente la table des adresses des fonctions BASIC.

Premières applications :

Dans un programme en langage machine, un JSR \$35EB effacera l'écran ; un JSR \$35E6 génèrera un "bip".

Un JSR \$5F2 permettra d'appeler un sous-programme écrit en BASIC à partir d'un programme en langage machine.

On pourra enfin étudier le traitement des diverses instructions du BASIC en listant les routines correspondantes.

IV. Le traitement des instructions

1. Utilisation de la table des adresses

Il nous faut trouver l'endroit où s'effectue le branchement aux différentes adresses de traitement des instructions.

On vient de voir que le BASIC doit pour celà, à partir du code C d'une instruction, calculer l'adresse :

$$A = \&H26B + (C - \&H80) * 2,$$

où se trouve l'adresse du sous-programme de traitement de l'instruction.

On va donc chercher s'il existe dans la ROM ou la RAM une valeur $\&H26B$ (ou à défaut une valeur de $\&H267$ à $26F$ par exemple, puisque la formule exacte du calcul de A est encore inconnue); cette valeur pourra être précédée d'un code d'instruction à adressage immédiat (ce ne pourra être que LDr ou ADDD) ou indexé, le pré-octet indiquant alors un déplacement sur 16 bits.

On reprend donc le programme 4; on obtient les deux adresses $\$38AC$ et $\$6204$, les deux octets $\$38AB$ et $\$6203$ contenant la valeur $\&H92$ (code de SBCA direct); ceci ne correspond pas à la condition ci-dessus.

Il faut donc chercher une instruction ayant $\$38AC$ ou $\$6204$ comme partie adresse; la page 0 du BASIC n'étant en effet située ni en $\$38$, ni en $\$62$ (voir paragraphe suivant), l'adresse ne peut être $\$AC$ ou $\$04$.

On reprend donc le programme 4 pour chercher $\$38AC$, puis $\$6204$; on ne trouve pas $\$38AC$, par contre on trouve $\$6204$ en $\$2B36$; l'octet $\$2B35$ contient $\&HBE$, c'est-à-dire le code de LDX en adressage étendu, ce qui correspond bien à ce que l'on cherche.

Remarquons que si l'on n'avait pas trouvé, on ferait comme au paragraphe précédent, c'est-à-dire que l'on chercherait une instruction LDr # valeur, avec "valeur" située par exemple entre $\$61F0$ et $620F$.

On peut donc maintenant à l'aide du désassembleur examiner les instructions autour de $\$2B35$; on y trouve effectivement le calcul de A ci-dessus.

Le listing complet est le suivant :

2B25	JSR	\$6270	Contient RTS
	BNE	\$2B2B	
	RTS		Si 3A en tête (')
2B2B	CMPA	# \$80	Mot BASIC?
	LBLO	\$722	Caractère ASCII (affectation)
	CMPA	# \$B9	Code de PLAY
	BHI	\$2B42	Code > &HB9
2B35	LDX	\$6204	Début table (&H26B)
	LSLA		(Code-&H80) * 2 → A
	TFR	A,B	
	ABX		(Code-80) * 2 + \$26B → X
	LDX	,X	Adresse traitement
	JSR	\$B2	Voir ci-après
	JMP	,X	Traitement de l'instruction
2B42	CMPA	# \$FF	
	BEQ	\$2B4E	
	CMPA	# \$D5	Code de <
	BLS	\$2AFF	Contient JMP \$7F3 (SN Error)
	JMP	[\$620E]	Contient \$7F3 (SN Error)
2B4E	JSR	\$B2	Voir ci-après
	CMPA	# \$9C	Code de MID\$
	LBEQ	\$1100	
	CMPA	# \$A1	Code de INPUT
	LBEQ	\$27A5	
	CMPA	# \$A4	Codé de SCREEN
	LBEQ	\$33CC	
	JMP	\$6273	Contient RTS

Application immédiate :

En modifiant le contenu des octets \$6204 et 6205, on pourra faire utiliser par le BASIC sa propre table d'adresses.

On pourra alors créer son propre langage (par exemple un BASIC "entier" très rapide) en écrivant les sous-programmes correspondants.

L'avantage est que l'on profitera ainsi de toute la partie de l'interpréteur réalisant le codage et l'édition.

2. Routine \$B2

Cette routine fondamentale permet le "balayage" des caractères d'un programme.

Le programme suivant, appelé par USR, nous donnera tout d'abord l'emplacement de la page 0 du BASIC :

TFR	DP,B	Registre de page 0 → B
STB	3,X	Poids faible accu. entier
RTS		

Ceci correspond à la suite des codes hexadécimaux 1F(31), B9 (code de PLAY), E7, 3, 39 (code de 9), qui seront donc implantés en mémoire (en \$65F8) le plus simplement possible par l'instruction 31 du programme suivant :

```
31 PLAYXX9$:POKE &H65FA,&HE7:POKE &H65FB,3
40 DEFUSR=&H65F8:PRINT HEX$(USR(0))
```

On obtient \$61, qui constitue l'emplacement de la page 0 du BASIC.

D'où le listing de la routine \$B2 :

61B2	INC	\$BA	
	BNE	\$61B8	
	INC	\$B9	
61B8	LDA	\$xxxx	Adresse caractère courant
	CPMA	# \$3A	Caractère ":" ?
	BHS	\$61C9	
	CPMA	# \$20	Espace ?
	BNE	\$61C5	
	JMP	\$B2	Élimine les espaces
61C5	SUBA	# \$30	Chiffre de 0 à 9 ?
	SUBA	# \$D0	
61C9	RTS		

Cette routine incrémente donc l'adresse du caractère courant, contenue dans \$61B9 et 61BA, puis retourne dans le registre A le code du caractère.

Le registre CC des codes conditions est positionné de la manière suivante :

- Z est mis à 1 si l'on a un 0 ou le caractère ":", c'est-à-dire une fin d'instruction ;
- C est mis à 1 si l'on a un chiffre, et à 0 dans le cas contraire.

REMARQUE: Cette routine existe sur tous les interpréteurs; elle peut être trouvée directement par le programme 1 (légèrement modifié) en cherchant l'endroit de la RAM où se trouve l'adresse du caractère courant du programme.

3. Contrôles de validité

La syntaxe des instructions, ou les valeurs de certains paramètres, sont systématiquement contrôlés par l'interpréteur au fur et à mesure de l'exécution d'un programme (par exemple ici, \$2B42 contrôle la validité d'un code).

Si l'un de ces contrôles est positif, il y a branchement en \$353, le registre B contenant le code de l'erreur: voir \$7F3 par exemple.

Les variables systèmes ERR (code de l'erreur) et ERL (numéro de la ligne où s'est produite l'erreur), situées respectivement en \$6189 et \$618A (voir chapitre II la routine \$770, en \$7C2) sont alors positionnées.

L'exécution est ensuite soit interrompue et le message d'erreur (contenu dans une table située en \$1722) affiché, soit poursuivie en α (jusqu'à l'instruction RESUME) s'il existe dans le programme une instruction ON ERROR GO TO α .

V. Boucle d'exécution d'un programme

Pour la trouver, il nous faut chercher dans la ROM un JSR, BSR ou LBSR \$2B25 (ou \$2B2B).

Or, en remontant avant le sous-programme de traitement des instructions, on trouve immédiatement en \$2B21 les instructions:

2B21	BSR	\$2B25	Traitement
	BRA	\$2AED	Début de la boucle

D'où le listing de la boucle:

2AED	STS	\$8C	
	JSR	\$32BB	Surveillance du clavier
	LDX	\$B9	Adresse caractère courant
	STX	\$34	
	LDA	,X+	Ø ou "':"?
	BEQ	\$2B02	Nouvelle ligne
	CMPA	# \$3A	
	BEQ	\$2B1F	
	JMP	\$7F3	SN Error

2B02	LDD	,X++	A-t-on trois 0?
	BEQ	\$2B65	Fin du programme
	LDD	,X+	Numéro de la ligne
	STD	\$2C	
	STX	\$B9	Sur dernier octet de l'en-tête
	LDA	\$86	≠ 0 si TRON (voir \$139E)
	BEQ	\$2B1F	Pas de trace demandée
	LDA	# \$5B	Code de "["
	JSR	\$1055	Écrit sur l'écran
	LDA	\$2C	Numéro de la ligne
	JSR	\$1ED1	Écrit le numéro
	LDA	# \$5D	Code de "]"
	JSR	\$1055	Écriture
2B11	JSR	\$B2	1 ^{er} octet de l'instruction
	BSR	\$2B25	Traitement
	BRA	\$2AED	Instruction suivante

Routine de surveillance du clavier:

32BB	JSR	\$6294	Contient RTS
	JSR	\$E809	KTST\$ du moniteur
	BCC	\$32BA	RTS (pas de touche)
	PSHS	B	Sauvegarde
	JSR	\$E806	GETC\$ du moniteur
32C8 à ...			Arrêt si CNT/C; boucle si STOP;
...32E1			Code touche → \$65B1 sinon

Application immédiate:

En intervenant en \$6294, on pourra supprimer ou modifier l'action du clavier: voir 4^e partie.

2

Le traitement des variables

Le traitement des variables sera décodé en étudiant l'instruction d'affectation; on trouvera ainsi du même coup le traitement des opérateurs et des fonctions BASIC.

Avant d'étudier la routine correspondante, il est nécessaire de connaître la représentation mémoire des variables et des tableaux, et aussi la manière dont est gérée la mémoire.

1. Représentation des variables

Les variables rencontrées lors de l'exécution d'un programme sont placées au fur et à mesure dans une zone de mémoire située après le programme lui-même; cette zone sera donc examinée à l'aide du programme 2.

Pour les TO7, on trouve que chaque nom de variable est précédé d'un octet contenant la valeur du type de la variable (dans les 4 bits de plus fort poids), puis le nombre de caractères du nom diminué de un (les %, \$, ! ou # éventuels ne comptent pas); on trouve ensuite le nom codé en ASCII, et enfin la valeur contenue dans un nombre d'octets égal à la valeur du type.

Exemple: on écrit dans le programme 2 :

```
110 AB=1.5:CDE%=-3:AB$="chaîne":CBA=-2.25:ADR0=&H669F
```

On obtient :

669F	0	0	0	<u>41</u>	41	42	81	40	0	0	<u>22</u>	43
66AB	44	45	FF	<u>FD</u>	<u>31</u>	41	42	6	66	D	<u>42</u>	43
66B7	42	41	82	90	0	0	<u>43</u>	41	44	52	<u>30</u>	8F

- Le bit de plus fort poids d'une variable entière (type égal à 2) représente le signe ; les 15 autres bits contiennent la valeur en complément à deux ;
- Une variable réelle simple précision (type égal à 4) est codée en mode virgule flottante : le premier octet est égal à l'exposant de 2 augmenté de &H80 ; les trois autres octets contiennent la mantisse normalisée (inférieure à 1 et supérieure ou égale à 0,5), le bit de plus fort poids étant remplacé par le signe de la valeur.

Exemple :

$$-2,25 = -4 \times 0,5625 = -2^2 (2^{-1} + 2^{-4})$$

d'où le codage :

82 90 00 00

- Une variable double précision (type égal à 8) est codée de la même manière, la mantisse occupant 7 octets.
- Le premier octet d'une variable chaîne (type égal à 3) contient le nombre de caractères de la chaîne ; les deux autres octets représentent l'adresse de la chaîne, située soit dans le programme lui-même en cas d'affectation simple (variable = "chaîne"), soit en fin de mémoire en cas de concaténation.

REMARQUE : Le nombre d'octets d'une variable réelle peut changer selon les ordinateurs ; de même, la représentation des noms de variable peut être différente. Par exemple, lorsque le nom est limité à deux caractères, le codage se fait toujours sur 2 octets, dont le bit de plus fort poids est positionné pour indiquer le type.

II. Représentation des tableaux

Les tableaux sont placés dans une zone située après celle des variables ; un tableau est créé dans cette zone par l'instruction DIM, ou à défaut par la première utilisation du tableau (la taille étant alors égale à 11).

Le nom d'un tableau est codé comme celui d'une variable ; il est suivi de l'en-tête suivant, qui précède les valeurs :

- deux octets contiennent le nombre total d'octets occupé par le tableau (y compris l'en-tête) ;
- un octet contient le nombre d'indices ;
- pour chaque indice, on a ensuite deux octets contenant la valeur maximale augmentée de un, en commençant par le dernier indice.

Pour les valeurs, c'est le premier indice qui varie d'abord, puis le second est incrémenté, etc...

Exemple : On écrit dans le programme 2 :

```
110 DIM AB%(3,2):AB%(1,0)=1:AB%(2,0)=2:AB%(3,0)=3:AB%(0,1)=4:AB%(1,1)=5:ADR0=&H66D3
```

On obtient :

66D3	21	41	42	0	1F	2	0	3	0	4	0	0
66DF	0	1	0	2	0	3	0	4	0	5	0	0

III. Gestion de la mémoire

L'interpréteur gère la mémoire à partir des adresses de chaque zone.

Les mémoires contenant ces adresses peuvent être trouvées soit par le programme 1 (on recherchera par exemple la mémoire contenant l'adresse du début de la zone des variables, trouvée par comptage à partir de \$65F5), soit par observation (le programme 2 permettra de trouver l'emplacement des chaînes, le contenu de \$618C, 618D donne la valeur du

pointeur S, etc...) soit par l'étude de l'instruction d'affectation (voir par exemple la routine \$A48, en \$A8A).

On trouve que ces mémoires sont situées en page 0 du BASIC ; contenant des adresses, elles occupent deux octets chacune :

- \$611C pointe sur la première instruction du programme, c'est-à-dire sur \$65F5 ;
- \$611E pointe sur le premier octet de la zone des variables ;
- \$6120 pointe sur le premier octet de la zone des tableaux ;
- \$6122 pointe sur le premier octet libre situé après les tableaux, c'est-à-dire sur le dernier octet utilisable par la pile S ;
- \$6124 pointe sur le "fond" de la pile S ; la zone des chaînes commence juste après ;
- \$612A pointe sur le dernier octet de la zone des chaînes ; les caractères utilisateurs éventuels sont situés juste après, en commençant par la ligne du bas du dernier ; la ligne du haut de GR\$(0) est située en FIN-1, FIN étant la plus haute adresse du BASIC (deuxième paramètre d'un CLEAR, ou plus haute adresse de la RAM par défaut).

IV. Recherche d'une variable ou d'un tableau

L'adresse de cette routine fondamentale de l'interpréteur sera trouvée en listant les premières instructions du traitement de l'affectation ; celle-ci doit en effet d'abord déterminer l'adresse où devra être rangé le résultat, qui sera ensuite calculé.

La routine recherche une variable ou un élément de tableau dans la zone de mémoire correspondante ; elle crée cette variable (sauf si elle est située dans un calcul d'expression, ceci pour le cas où la valeur devrait être affectée à un élément de tableau, non déplaçable pendant l'affectation elle-même) ou le tableau lui-même, en les initialisant à 0 s'ils n'existent pas encore ; elle retourne enfin l'adresse de la valeur correspondante.

Dans le cas des TO7, la routine est située en \$A48 (on trouve en effet en \$722 l'instruction JSR \$A48) ; elle retourne l'adresse de la valeur dans le registre X et dans la mémoire \$613D ; le type de la valeur est placé dans l'octet d'adresse \$6105 ; enfin, \$61B9 (contenant l'adresse du caractère courant du programme) est positionné sur le premier caractère qui suit la variable ou l'élément de tableau.

Le listing est en effet le suivant :

A48	CLRB		
	JSR	\$B8	1 ^{er} caractère du nom à chercher
A4B	STB	\$04	Entrée pour DIM, avec B≠0
	JSR	\$6297	Contient RTS
	BSR	\$A0A	Nom rangé à partir de \$657A ; nombre de caractères en \$613C
A52 à	...A75		Valeur du type → \$6105
A76	JSR	\$B2	1 ^{er} caractère après le nom
	LDB	\$07	Contient normalement 0
	DECB		
	LBEQ	\$B64	Recherche début d'un tableau
	INCB		
	BNE	\$A88	Pas de tableau (cas de FOR)
	SUBA	#\$28	Code de "("
	LBEQ	\$B12	Élément de tableau
A88	CLR	\$07	Variable
	LDX	\$1E	Début de zone des variables
A8C	CMPX	\$20	Fin zone des variables ?
	BEQ	\$AA2	Variable n'existe pas encore
	LDB	,X	1 ^{er} octet d'un variable
	LSRB		
	LSRB		
	LSRB		
	LSRB		
	PSHS	B	Garde les 4 bits de fort poids
	JSR	\$ADB	Valeur du type
	PULS	B	Comparaison des noms
	BEQ	\$AD8	Variable trouvée
	ABX		Pas la bonne variable
	BRA	\$A8C	Variable suivante
AA2 à	...		Variable ajoutée en fin de zone (sauf si \$A48 a été
...AD7			appelée en \$800), après déplacement de la zone
			des tableaux
AD8	STX	\$3D	Adresse dans X et \$3D
	RTS		

Puis on a, pour les tableaux :

B12 à	...		
...B62			Calcule et empile les indices
B63	LDB	#\$5F	Ou CLRB en \$B64
B65	PSHS	B	5F ou 0 dans la pile
B67 à	...		Cherche nom du tableau (et RTS si pile contient
...BE8			0) ; tabl. créé par \$B97 si n'existe pas encore
BE9 à			
...C12			Calcule l'adresse de l'élément
C13	STX	\$3D	Adresse dans X et \$3D
	RTS		

Le lecteur est bien entendu vivement invité à étudier les instructions ou sous-programme non listés ici pour des raisons de place.

Amélioration possible : Lorsqu'on étudie le traitement des instructions du BASIC, on s'aperçoit que l'interpréteur appelle la routine \$A48 chaque fois qu'il rencontre une variable ou un élément de tableau.

Par exemple dans le cas d'une boucle, il y a donc à *chaque passage* exploration de la RAM jusqu'à trouver la valeur correspondante, située pourtant toujours au même endroit dans le cas d'une variable.

On verra dans la 4^e partie que ceci peut être évité, en intervenant en \$6297 ; la vitesse d'exécution des programmes sera alors très nettement augmentée.

V. Traitement d'une expression

1. Instruction d'affectation

L'instruction se présente sous la forme :

variable = expression

("variable" pouvant être un élément de tableau).

La routine de traitement est la suivante :

722	JSR	\$A48	Recherche de la variable
	STX	\$3F	Adresse de la valeur
	LDB	#\$D4	Code de "="
	JSR	\$D0	Contient JMP \$7EB
	LDA	\$05	Type de la variable
	PSHS	A	
	JSR	\$81A	Calcul de l'expression
	PULS	A	
734	JSR	\$2510	Conversion éventuelle de l'exp. dans le type de la variable
	JSR	\$CD	Contient JMP \$2502
	LBNE	\$1C36	Si type 2,4 ou 8 ; range valeur à l'adresse contenue dans \$3F
73D à ...			Si type 3 ; affecte la chaîne résultat à la variable
...76A			

Les routines situées en \$7EB et \$2502 sont appelées de nombreuses fois par l'interpréteur.

- \$7EB est utilisée pour tous les contrôles de syntaxe; elle teste si le caractère courant est bien égal à celui contenu dans l'accumulateur B (SN Error sinon), puis elle retourne le caractère suivant; on a en effet :

7EB	CMPB	[\$61B9]	Code du caractère courant
	BNE	\$7F3	
	JMP	\$B2	Caractère suivant
7F3	LDB	#\$02	Code erreur SN: syntax error
	JMP	\$353	Affichage erreur; arrêt

- \$2502 positionne le registre CC du 6809 selon la valeur x d'un type, contenue dans l'octet \$6105 (ou dans le registre A si le point d'entrée est en \$2504):

Si x = 2 (entier), N et C sont mis à 1, Z et V à 0
 Si x = 3 (chaîne), Z et C sont mis à 1, N et V à 0
 Si x = 4 (réel), V et C sont mis à 1, N et Z à 0
 Si x = 8 (double précision), N,Z,V et C sont mis à 0.

On a en effet :

2502	LDA	\$05	
2504	CMPA	#\$08	0 → C si 8
	DECA		
	DECA		
	DECA		1 → Z si 3; 1 → N si 2
	BLE	\$250F	Si 2 ou 3
	BCC	\$250F	Si 8
	ORCC	#\$02	Si 4; 1 → V
250F	RTS		

2. Calcul d'une expression

Il est effectué par la routine \$81A; celle-ci range le résultat x dans "l'accumulateur flottant" (utilisé par l'instruction USR du BASIC) situé à partir de \$6155.

- Si x est entier, il est placé en \$6157 et 6158;
- Si x est une chaîne, l'adresse du descripteur de la chaîne (c'est-à-dire toujours \$658A, adresse où l'on trouve la longueur et l'adresse de la chaîne) est placée encore en \$6157 et 6158;

- Si x est un réel, la valeur absolue est rangée de \$6155 à 6158, le signe étant contenu dans \$615D (dans le bit de plus fort poids);
- Si x est un réel double précision, l'accumulateur va de \$6155 à 615C; le signe est toujours en \$615D.

Enfin, le type du résultat est placé dans \$6105, et il y a positionnement de \$61B9 sur le premier caractère qui suit l'expression.

Le listing est le suivant:

81A	BSR	\$815	Revient sur caractère précédent
	CLRA		
81D	CMPX	# \$9641	Ou LDA \$41 en \$81E
	PSHS	A	
	LDB	# \$01	
	JSR	\$336	Test de débordement mémoire
	JSR	\$770	Traite un opérande; valeur dans l'accumulateur flottant
82A à	...875		Décodage d'un opérateur ou RTS
876	BSR	\$883	Calcule une sous-expression
878	BRA	\$82A	Suite de l'expression

3. Traitement d'un opérande

On verra que la routine \$883 contient un JSR \$81E pour chercher la valeur de l'opération situé après l'opérateur T; tous les opérandes sont donc traités par la routine suivante, située en \$770:

770	JSR	\$627C	Contient RTS
	LDX	\$B9	
	JSR	\$B2	1 ^{er} caractère de l'opérande
	BEQ	\$76B	MO Error (Missing Opérand)
	BCC	\$780	On n'a pas un chiffre
77B	STX	\$B9	On a un chiffre
	JMP	\$1E03	Traitement des constantes
780	JSR	\$A36	Retourne C=0 si on a une lettre
	BCC	\$800	Variable
	CMPA	# \$2E	Code de "."
	BEQ	\$77B	Constante réelle
789 à	...		Décode et traite \pm unaires, " (chaîne), NOT, &
...7D4			ERR et ERL, USR
7D5	CMPA	# \$BD	Code FN
	LBEQ	\$623C	Contient JMP \$7F3 (SN Error)

	INCA		A-t-on FF (fonction)?
	BEQ	\$80D	Contient JMP \$2A93
7DE à ...7F7	...		Teste si on a "(" et SN Error sinon ; calcule l'exp. entre ()
800	JSR	\$A48	Recherche de la variable
	STX	\$57	Adresse
	JSR	\$CD	Contient JMP \$2502
	BNE	\$80A	Si variable numérique
	RTS		Si variable chaîne
80A	JMP	\$1C02	Valeur dans accu. flottant

4. Applications

En intervenant en \$623C, on pourra utiliser des fonctions utilisateurs (FN) : voir troisième partie.

En intervenant en \$627C, on pourra modifier la routine \$770 du traitement d'un opérande, par exemple au niveau du calcul des constantes, ou de la recherche d'une variable : voir quatrième partie.

VI. Traitement des fonctions BASIC

On vient de voir qu'il est effectué en \$2A93 ; on a :

2A93	JSR	\$6288	Contient RTS
	JSR	\$B2	Deuxième octet du code
	TFR	A,B	Code C → B
	LSLB		$x = (C - \$80) * 2 \rightarrow B$
	JSR	\$B2	Caractère suivant
	CMPB	≠ \$4C	PTRIG
	BLS	\$2AA5	Si inférieur ou égal
	JMP	[\$6213]	Contient \$7F3 (SN Error)
2AA5	PSHS	B	x → pile
2AA7 à ...2AE1			Traitement des paramètres ; conversion en réel pour les fonctions de code \$FF84 à 89 (SQR à TAN)
2AE2	PULS	B	Dépile x
	LDX	\$6209	Contient \$0020
	ABX		X pointe sur x ^e adresse
	JSR	[0,X]	Traitement de la fonction
	JMP	\$24FD	Contrôle du type

On constate immédiatement que la table des fonctions commence en \$0020, ce qui correspond à ce que l'on avait trouvé au chapitre I ; elle se termine en \$006D.

On trouvera en annexe les adresses de traitement de toutes les fonctions BASIC (lues dans la table, ou décodées différemment, par exemple pour TAB ou SPC, etc...).

Application immédiate : Un programme écrit en langage machine peut utiliser directement des fonctions BASIC, par exemple SQR, CSNG, etc... : voir 3^e partie.

VII. Traitement des opérateurs BASIC

L'enchaînement des routines correspondantes est relativement complexe.

1. Table des opérateurs

On a vu que les opérateurs sont décodés en \$82A ; on a :

82A à 833			Initialisations
834 à ...84A			Codage dans \$6143 des opérat. de relation : 1 si > , 2 si = , 3 si ≥ , 4 si < , 5 si ≠ , 6 si ≤
84B	LDB	\$43	
	LBNE	\$97A	Traite les opérat. de relation
851 à ...85E			x = (code opérat. - \$C7) → B, et test de concaténation de chaîne
85F	LBEQ	\$DBB	Traitement de la concaténation
863	PSHS	B	x → pile
	LSLB		2 * x → B
	ADDB	,S,	3 * x → B et dépilement de x
	LDX	#\$006E	Début de la table
	ABX		\$6E + 3x → X
86C à ...875			Voir ci-après

La table des opérateurs commence donc en \$006E ; elle comprend trois octets pour chaque opérateur :

- le premier représente la priorité de l'opérateur (&H7F pour la plus grande, celle de la puissance) ;

— les deux autres l'adresse de traitement.

Les informations concernant tous les opérateurs sont listées en annexe.

2. La routine \$883

Elle calcule une sous-expression aTb; le listing est le suivant :

883	STB	\$41	Priorité de l'opérat. courant T
	CMPB	# \$7F	
	BEQ	\$8F3	Opérateur puissance
	LDU	1,X	
	PSHS	U	Adresse traitement de T → pile
	CMPB	# \$51	
	BLO	\$905	Opérateur logique (AND à IMP)
	ANDB	# \$FE	
	CMPB	# \$7A	
	BEQ	\$90E	Opérateur MOD et @
	JSR	\$939	Opérande a → pile
	JSR	\$81E	Calcule b; bit le T' suivant
89D	JSR	\$24FD	b numérique (SN Error sinon)?
	JSR	\$95A	Dépilement dans \$6163 à 616B de l'opérande en haut de la pile (on y trouve donc maintenant l'adresse du traitement de T)
8A3 à			Conversion si a et b de types ≠; permutation s'ils sont entiers
... 8B1			
8B2	RTS		Branche au traitement de T

On observera qu'en \$8F3 (opérateur puissance) et \$905 (opérateurs logiques), on a exactement la même succession d'appels des routines \$939, \$81E et \$95A, suivie du branchement à la routine de traitement de l'opérateur (\$2391 pour ↑ ; l'adresse située en \$7B sert en effet pour les opérateurs de relation: voir \$97A à 99B, puis \$99C).

Le lecteur observera aussi le dépilement dans Y des adresses de retour des routines \$939 et \$95A, le retour se faisant alors par une instruction JMP ,Y; ceci permet de n'empiler que les opérandes et opérateurs successifs.

3. La gestion des priorités

On constate que \$883 appelle lui-même la routine \$81E; celle-ci range donc la priorité de T (opérateur courant) dans la pile, puis lit l'opérande b et l'opérateur T' suivants; on arrive alors en \$86C, où l'on a :

86C	LDA	,S	Priorité de T, rangée en \$81E
	CMPA	,X	Priorité de T', dans la table
	BHS	\$853	Si priorité de T \geq celle de T'
	BSR	\$812	b numérique (SN Error sinon) ?
	BSR	\$883	

\$853 contient l'instruction **PULS A,PC**.

Donc si la priorité de T est supérieure ou égale à celle de T', on dépile la priorité de T et on retourne en \$89D (car le JSR \$81E en \$89A place \$89D dans la pile); d'où le calcul de l'opération correspondant à T (cas de $a * b + \dots$).

Si la priorité de T est inférieure à celle de T', il y a un nouvel appel de \$883, c'est-à-dire que la priorité et l'adresse du traitement de T sont empilés, ainsi que la valeur de l'opérande b; l'opération correspondante ne sera donc exécutée que plus tard (cas de $a + b * \dots$).

4. Applications

Un programme écrit en langage machine peut utiliser les routines de traitement des opérateurs BASIC, en particulier *, @ et MOD : voir la troisième partie.

On pourra aussi écrire une routine simplifiée de traitement des opérateurs (pour + par exemple) utilisable en BASIC et conduisant à des calculs plus rapides de 50 % : voir la troisième partie (INC).

3

Etude de quelques instructions BASIC

Nous détaillons ici le fonctionnement de certaines instructions fondamentales du BASIC TO7.

Les instructions sont bien sûr traitées sous des formes très voisines sur tous les micro-ordinateurs, d'où l'intérêt général de cette étude.

Celle-ci nous permettra d'envisager diverses interventions sur l'interpréteur, que nous verrons dans la suite de notre ouvrage.

1. Instruction de branchement

Le mot GO est traité en \$606 ; selon qu'il est suivi de TO ou de SUB, on va respectivement en \$624 ou en \$612, avec \$61B9 positionné sur le premier caractère suivant.

1. GO TO

L'instruction GO TO α est donc traitée de la manière suivante :

624	JSR	\$B8	Premier chiffre de α
	JSR	\$6FD	Calcul de α , rangé en \$6130
	BSR	\$66E	Cherche le 0 de fin de ligne
	LEAX	1,X	X pointe sur la ligne suivante
	LDD	\$30	$\alpha \rightarrow D$
	CMPD	\$2C	n° β de la ligne du GO TO
	BHI	\$636	$\alpha > \beta \Rightarrow \alpha$ après la ligne actuelle ; sinon, chercher avant
	LDX	\$1C	Adresse du début du programme
636	JSR	\$4A4	Retourne dans X l'adresse de la ligne de n° α (ou C=1 si n'existe pas)
	BCS	\$655	UL Error
	LEAX	-1,X	Adresse du 0 précédant α
	STX	\$B9	Positionne le caractère courant
	RTS		Provoque l'exécution de α

Routine \$66E (ou \$66B) : Elle commence par :

66B	LDB	# \$3A	Code de " : "
66D	LDA	# \$5F	Ou CLRB en \$66E

On a ensuite de \$66F à 696 la recherche, à partir du caractère courant du programme, à la fois d'un 0 et du caractère dont le code a été mis dans B (en \$66B ou \$66E) ; l'adresse correspondante est placée dans X.

\$66E recherche donc la fin de la ligne courante (0) ; \$66B recherche la fin de l'instruction en cours (0 ou \$3A).

Cette routine fondamentale est utilisée chaque fois que l'interpréteur doit sauter quelque chose ; elle est donc en particulier utilisée aussi par RETURN, REM, DATA, ON, IF et FOR.

Dans le cas des TO7, la routine détecte les caractères de code \$22 (guillemets ; les caractères de code \$3A sont alors ignorés jusqu'aux deuxièmes guillemets), FF (fonction ; un octet est alors sauté) et 89 (ceci pour les cas où l'on a des IF imbriqués ; la mémoire \$6171 est alors incrémentée).

Conséquence immédiate :

Si l'on intervient dans le programme lui-même, par exemple, pour remplacer des constantes par leur valeur binaire, ou des variables par leur adresse (voir quatrième partie), il ne faudra jamais placer les valeurs &H0, 22, 3A, 89 ou FF.

2. GOSUB et RETURN

Le traitement de GOSUB α est le suivant :

612	LDB	#\$03	On va empiler trois registres
	JSR	\$336	Test de débordement mémoire
	LDU	\$B9	Caractère courant
	LDX	\$2C	Numéro β de la ligne courante
	LDA	#\$BC	Code de SUB
	PSHS	U,X,A	
	BSR	\$624	GO TO : positionne sur ligne α
	JMP	\$2AED	Boucle d'exécution du programme

Lorsque l'interpréteur rencontre RETURN (traité en \$640), il positionne S sur la valeur en haut de la pile (RG Error si ce n'est pas &HBC), puis arrive en \$65D où l'on trouve :

65D	PULS	A,X,U	
	STX	\$2C	Restauré β (ligne du GOSUB)
	STU	\$B9	Caractère courant sur α
663	BSR	\$66B	Cherche fin de GOSUB α
	CMPX	#\$8D06	Ignoré ici (sert pour REM)
	STX	\$B9	Caractère courant
66A	RTS		On est revenu après GOSUB

REMARQUE: Les boucles non terminées sont dépilées en \$646 (routine \$2F3 appelée avec &HFF dans \$613F: voir paragraphe III), avant le retour au programme principal).

3. ON

L'instruction ON expression GO... est traitée en \$6D7 (voir \$36B5), où la valeur x de l'expression est rangée dans \$6158; l'interpréteur se positionne alors sur le x^e numéro de la liste (lu et calculé par la routine \$6FD) et se branche en \$608, c'est-à-dire au traitement de GOTO et GOSUB.

4. Amélioration possible

A chaque instruction de branchement rencontrée, il y a calcul de l'étiquette et exploration du programme; ceci est refait chaque fois si l'on repasse sur le même branchement.

On verra dans la quatrième partie que ceci peut être évité, ce qui améliore nettement la vitesse d'exécution.

II. Instruction de test

L'instruction IF expression... est traitée en \$697, où l'on trouve :

697	JSR	\$81A	Calcul de l'expression
69A	à 6AF		SN Error si pas THEN ou GOTO
6B0	JSR	\$1CC8	1 → Z si exp. fausse (0), 0 si vraie
	BNE	\$6C8	Exécute le GOTO ou la suite du THEN ; sinon, doit être sauté
	CLR	\$71	Va compter nbre de IF imbriqués
6B7	BSR	\$663	Cherche 0 ou \$3A (voir RETURN)
	TSTA		0 ou 3A ?
	BEQ	\$66A	Pas ELSE ⇒ ligne suivante (RTS)
	JSR	\$B2	\$3A ; ELSE ou suite du THEN ?
	CMPL	≠\$8F	Code de ELSE
	BNE	\$6B7	Suite du THEN
	DEC	\$71	A-t-on des IF imbriqués ?
	BPL	\$6B7	Pas encore le bon ELSE
	JSR	\$B2	Trouvé
6C8	JSR	\$B8	Caractère courant
	LBCS	\$624	Chiffre ⇒ branchement (GOTO)
	JMP	\$2B25	Traitement des instructions

REMARQUE: Lors de la saisie d'un programme, ELSE est codé par &H8F précédé de &H3A, d'où l'utilisation de la routine \$66B (par BSR \$663).

III. Instruction FOR... NEXT

1. Traitement de FOR

Il est effectué en \$1578 ; après avoir initialisé la variable de contrôle, la routine empile successivement :

- l'adresse du 0 ou &H3A terminant l'instruction FOR (2 octets),
- le numéro (2 octets) de la ligne courante (grâce à ces deux informations, on pourra se repositionner après le NEXT sur la première instruction de la boucle),
- la valeur finale de la variable de contrôle (4 octets),
- le signe du pas d'incréméntation (1 octet),

- la valeur du pas (4 octets)
- le type du pas (1 octet, égal à 2 ou 4),
- l'adresse du NEXT (2 octets) correspondant au FOR (trouvé par la routine \$16AD décrite ci-après),
- l'adresse de la variable de contrôle (2 octets),
- la valeur &H81, c'est-à-dire le code de FOR (1 octet).

La routine positionne ensuite \$612C avec le numéro de la ligne du NEXT (\$61B9 est déjà positionné sur le NEXT, par la routine \$16AD) et appelle en \$1602 la routine \$1604, correspondant au traitement de NEXT sans incrémentation de la variable de contrôle (grâce à la mémoire \$6185 initialisée ici à &H4F et non à 0).

L'exécution se poursuit donc soit après le NEXT, soit à partir de la première instruction de la boucle.

REMARQUE: Cet appel de la routine du traitement de NEXT (sans incrémentation) n'est pas effectué par tous les BASIC; la boucle est alors toujours exécutée au moins une fois.

2. Traitement de NEXT

NEXT est traité en \$1605, où la mémoire \$6185 est initialisée à 0.

Puis en \$161A est appelée la routine \$2F3, qui balaye la pile S à partir du sommet pour trouver le FOR correspondant au NEXT; le BASIC Microsoft autorisant en effet les sorties anormales de boucle par GOTO, le bon FOR peut ne pas être situé au sommet de la pile.

Le registre S est alors modifié pour pointer sur la valeur &H81 correspondant à ce FOR, ce qui dépile les boucles dont on est sorti anormalement; puis les valeurs empilées sont récupérées.

La variable de contrôle est alors incrémentée (sauf si \$6181 est différent de 0) et le test de fin de boucle exécuté.

S'il est vrai, le numéro de la ligne du FOR est récupéré (en \$1622, par LDX 15,S et STX \$2C), puis l'adresse de la première instruction de la boucle (par LDX 17,S et STX \$B9); il y a enfin branchement en \$2AED, c'est-à-dire à la boucle d'exécution du programme; l'instruction exécutée est donc la première après le FOR.

Si le test est faux, les 19 octets empilés par le FOR sont dépilés (en \$166E) et il y a encore branchement en \$2AED; l'exécution se poursuit donc après le NEXT.

3. La routine \$16AD

Il s'agit d'une deuxième routine d'exploration du programme (la première étant \$66B ou \$66E) ; son point d'entrée est en fait soit en \$16AC (\$6170 est alors initialisé par &H4F), soit en \$16AD qui initialise \$6170 à 0.

Le programme est ici exploré à partir du caractère courant jusqu'à trouver un octet contenant 0, &H3A (":"), &H8F (ELSE) ou &HC4 (THEN).

Selon la valeur de \$6170, la routine teste si le caractère suivant est FOR (&H81) ou WHILE (&HAF) ; si c'est le cas, on a deux boucles imbriquées ; le registre B est alors incrémenté et il y a retour au début.

Si l'on a un NEXT (&H82) ou un WEND (&HB0), toujours selon la valeur de \$6170, la routine examine si c'est le bon ; pour cela, B est décrémenté et il y a retour au début si l'on n'a pas 0.

Si c'est le cas, la routine retourne le numéro de la ligne dans \$6178 ; le caractère courant est positionné sur le NEXT ou le WEND.

Conséquence immédiate :

Si l'on intervient dans le programme lui-même, on ne devra jamais introduire des valeurs &H3A81, ou &H8F81, ou &HC4AF, etc...

4. Applications

La routine \$16AC trouve le WEND correspondant à un WHILE.

Il sera donc très facile de créer les routines complètes de traitement de WHILE et WEND : voir 3^e partie.

IV. Les instructions graphiques

1. Le graphisme des TO7

L'écran est divisé en 200 lignes de 40 segments de huit points ; chaque segment est décrit par deux octets de la mémoire écran.

Celle-ci est en effet composée de deux blocs de 8K-octets situés à la même adresse \$4000 ; ils sont sélectionnés par la valeur du bit 0 d'un port d'un PIA (port C du circuit 6846), lui même situé à l'adresse \$E7C3.

- la valeur 1 sélectionne la mémoire de "forme"; chaque bit d'un octet représente alors un point de l'écran, qui appartient soit à la forme (bit égal à 1), soit au fond (bit égal à 0).

En général, la forme est constituée par un message ou un graphisme.

Exemple :

```
100 POKE &HE7C3,PEEK(&HE7C3) OR 1:ADR=&H4000+C*8+L*40
110 POKE ADR,2^(7-C MOD 8)
```

affiche le point de coordonnées (C,L), C étant la colonne (0 à 319) et L la ligne (0 à 199).

REMARQUE: La routine \$F161 du moniteur du TO7 met à 1 le bit 0 de \$E7C3 (\$F328 pour le TO7-70).

- la valeur 0 sélectionne la mémoire de "couleur"; chaque octet représente alors la couleur de la forme (dans les bits 3,4,5) et du fond (bits 0,1,2) d'un segment de 8 points.

Exemple :

```
200 POKE &HE7C3,PEEK(&HE7C3) AND &HFE:POKE ADR,&H0F
```

affiche le point précédent en rouge (code 1=&B001), le fond du segment étant blanc (code 7=&B111).

REMARQUE 1: Sur le TO7-70, les bits 6 et 7 de la mémoire couleur sélectionnent respectivement la couleur pastel pour la forme et pour le fond s'ils sont mis à 0; ils sont toujours à 1 sur le TO7.

REMARQUE 2: \$E7C3 contrôle aussi entre autres l'affichage en majuscules ou minuscules (bit 3), et la couleur du cadre de l'écran (bits 4,5 et 6); voir annexes.

2. Line (x₁, y₁) - (x₂, y₂)

L'instruction trace une droite entre les points P₁ (x₁,y₁) (qui peut être omis) et P₂ (x₂,y₂); le traitement est le suivant:

34FB	BSR	\$354B	Affiche P ₁ ; P ₂ → X et Y
	LDX	\$6576	Abscisse de P ₂ (colonne)
3500	LDY	\$6578	Ordonnée de P ₂ (ligne)
3504	JMP	\$E80C	Tracé (DRAW\$ du moniteur)

Routine \$354B : elle est utilisée par les 4 instructions LINE, BOX, BOXF et PSET, quel que soit le mode (graphique ou caractère).

La routine place d'abord dans les registres X et Y les coordonnées du dernier point affiché ; elles sont remplacées par les coordonnées de P_1 si elles figurent dans l'instruction (routine \$34CB) ; X et Y sont alors placés respectivement en \$6572 et \$6574, puis rangés dans la pile.

Les coordonnées de P_2 sont ensuite lues (toujours par \$34CB), puis stockées en \$6576 et \$6578.

Si l'instruction opère en mode caractère, les attributs sont traités en \$3574, qui se termine par un JMP \$E833 affichant un caractère (routine CHPL\$ du moniteur).

Si l'on est en mode graphique, l'attribut éventuel couleur est traité en \$35C1 ; s'il ne figure pas dans l'instruction, la routine prend (en \$35B7) la couleur forme courante contenue dans le registre moniteur \$603B (COLOUR) ; dans les deux cas, la couleur est rangée dans le registre moniteur \$6038 (FORME).

X et Y sont enfin dépilés, ce qui restaure les coordonnées de P_1 , le registre \$6041 (CHDRAW) est mis à 0 et il y a branchement en \$E80F qui affiche le point graphique (routine PLOT\$ du moniteur).

3. BOX et BOXF (x_1, y_1) - (x_2, y_2)

L'instruction BOX est traitée en \$3507, où la mémoire \$6571 est initialisée à 0 ou à &H46 si l'instruction est suivie de la lettre F.

Il y a ensuite appel de la routine \$354B, qui range donc entre autres x_1 et y_1 dans X et Y ; selon la valeur de \$6571, il y a branchement en \$351A pour BOX et en \$352C pour BOXF ; on a alors :

351A	LDX	\$6576	$x_2 \rightarrow X$; on a $Y=y_1$
	BSR	\$3504	Trace drte. horiz. $P_1-(x_2, y_1)$
	BSR	\$3500	Trace drte. vert. $(x_2, y_1)-P_2$
	LDX	\$6572	$x_1 \rightarrow X$; on a $Y=y_2$
	BSR	\$3504	Trace drt. horiz. $P_2-(x_1, y_2)$
	LDY	\$6574	$y_1 \rightarrow Y$; on a $X=x_1$
352A	BRA	\$3504	Trace drt. $(x_1, y_2)-P_1$ et RTS

La routine \$352C (BOXF) opère en traçant la droite horizontale d'ordonnée y_1 , puis celle d'ordonnée $y_1 \pm 1$ (selon que y_2 est supérieur ou inférieur à y_1), etc..., jusqu'à arriver à y_2 .

Applications :

En s'inspirant des routines ci-dessus, on pourra facilement écrire des routines dessinant par exemple des losanges, ou même des cercles "pleins" : voir troisième partie.

4. PSET (x, y)

C'est l'instruction la plus simple ; elle est traitée en \$34EC, où l'on trouve :

34EC	BSR	\$34CB	$x \rightarrow X, y \rightarrow Y$
	BSR	\$34F1	
	RTS		
34F1	PSHS	X,Y	
	BRA	\$3565	Traite couleur ; affiche P

\$3565 appartient en effet à la routine \$354B.

Les autres instructions

L'étude accomplie jusqu'ici débouche comme nous allons le voir sur un grand nombre d'applications.

Il reste cependant bien d'autres instructions à étudier, que le lecteur choisira en fonction de son intérêt personnel ; il s'aidera pour cela des listes des principales routines et adresses du BASIC, données en annexe (listes non exhaustives !)

Par exemple, si l'on désire lever la protection d'un programme, on devra étudier le traitement de LIST (ou de PEEK ou POKE par exemple) ; on constatera alors que la protection est assurée par la routine \$2D29, qui teste la mémoire \$61A2.

La protection sera donc levée en chargeant (par LOADM, puisque POKE est interdit) la mémoire \$61A2 ; on devra pour cela enregistrer préalablement l'octet \$61A2 sur cassette (par SAVEM "nom", &H61A2, &H61A2, Ø).

4

Méthode pratique de décodage d'un interpréteur

Nous avons déjà largement expliqué, en particulier au chapitre I, la démarche à suivre pour décoder un interpréteur BASIC quelconque, à priori totalement hermétique !...

Nous résumons ici cette méthode, opérant avec 5 programmes BASIC de cinq ou six lignes chacun et un peu de réflexion...

1. Début d'un programme

Le programme 1 permet de trouver l'adresse B du début d'un programme, la recherche devant commencer à l'adresse A du début de la RAM (ligne 30 à modifier).

2. Codage d'un programme

Les octets situés à partir de B seront examinés grâce au programme 2 ; on déterminera ainsi le codage des instructions et de quelques mots-clés du BASIC.

3. Tables des mots-clés du BASIC

Le programme 1 permet de trouver dans le BASIC (ligne 30 à modifier) l'adresse C du début de la table des noms d'instructions, et l'adresse D du début de la table des noms de fonctions ; on cherchera pour cela (ligne 20 à modifier) les (n-1) premières lettres d'un mot-clé de n caractères situé vers le début de la table à localiser.

On en déduira les codes de tous les mots clés du BASIC, grâce au programme 3 (valeurs &H80, &HD5 et 86 de la ligne 240 à modifier, ainsi que la ligne 220).

4. Utilisation des tables de noms

Le programme 4 permet de trouver dans le BASIC et dans la RAM, en principe entre A et B (lignes 320 et 330 à modifier) les valeurs C et D.

S'il existe une adresse E de la RAM contenant C ou D, il suffira en principe de modifier les octets E et E+1 pour que le BASIC puisse utiliser un autre vocabulaire (à créer).

5. Tables des adresses

Le programme 5 permet de trouver dans le BASIC (ligne 410 à modifier) l'adresse F du début de la table des adresses des instructions.

On disposera alors déjà de toutes les adresses de traitement des instructions du BASIC, que l'on pourra donc étudier.

6. Recherche des octets contenant l'adresse de la table

Le programme 4 permet de trouver dans le BASIC et dans la RAM la valeur F (si on ne la trouve pas, on cherchera une valeur comprise entre F-d et F+d, avec d=10 par exemple) ; on obtient une ou plusieurs adresses G (\$38AC et \$6204 pour le TO7).

7. Routine de traitement des instructions

On examinera les octets situés "autour" de chaque adresse G trouvée à l'étape précédente.

S'ils contiennent pour une de ces adresses le calcul de : $x=(F \pm d) + (C-C0) * 2$ (d pouvant être éventuellement différent de 0, et C0 étant le plus petit code d'instruction BASIC, c'est-à-dire en principe toujours &H80), on a trouvé la routine de traitement des instructions ; on va donc directement à l'étape suivante.

Si l'on ne trouve pas le calcul précédent, le programme 4 permet de trouver la (ou les) valeur G précédentes dans la mémoire (ligne 310 à modifier) ; on obtient donc une ou plusieurs adresses H (si l'on ne trouve aucune adresse contenant G, on cherchera en modifiant la ligne 360 une valeur comprise entre G-1 et G+1).

Pour les T07 par exemple, on obtient la seule adresse \$2B36.

On doit forcément trouver alors "autour" d'une de ces adresses H le calcul de :

$$y=([G] \pm d) + (C-C0) * 2$$

ce qui correspond à la routine de traitement des instructions ([G] désigne le contenu de la mémoire d'adresse G, c'est-à-dire F).

8. Boucle d'exécution des programmes

Elle est en principe toujours située juste avant la routine de traitement des instructions (si ce n'est pas le cas, on cherchera dans le BASIC l'appel de cette routine, grâce au programme 1).

9. Les routines fondamentales

L'étude de l'instruction d'affectation donnera enfin l'adresse des routines de recherche d'une variable, de traitement d'un opérande (qui traite elle-même les fonctions) et du calcul d'une expression (où l'on trouvera le traitement des opérateurs).

Ceci terminera le décodage de l'interpréteur et ouvrira la porte aux applications...

Troisième partie

MODIFIER ET COMPLETER LE BASIC

Nous avons vu que les adresses des principales tables du BASIC TO7 sont situées dans la RAM, et donc modifiables.

De même, beaucoup de routines (\$A48, \$770, etc...) "passent" par la RAM, par exemple en appelant un sous-programme constitué d'un simple RTS (situé entre \$626D et \$62A8); ces routines pourront donc être déroutées.

On pourra donc appliquer facilement, bien sûr après adaptation, les modifications que nous allons envisager à tous les micro-ordinateurs dont l'interpréteur remplit les mêmes conditions, ce qui est le cas le plus fréquent; signalons d'ailleurs pour ceux dont le BASIC est entièrement figé en ROM que nous verrons dans la quatrième partie comment intervenir tout de même sur le fonctionnement de l'interpréteur.

Après avoir présenté tout d'abord une "récréation" consistant à remplacer le BASIC d'origine par un BASIC entièrement francisé, nous décrivons la réalisation de deux compléments classiques et utiles n'existant pas sur les TO7: conversions radiants-degrés et boucle WHILE-WEND.

Nous donnons ensuite une méthode générale permettant de créer de nouvelles instructions, que l'on pourra utiliser exactement comme les instructions initiales.

Nous appliquons tout d'abord ceci pour créer les deux instructions INC et SWAP d'incrémentation rapide et de permutation de variables.

Nous décrivons ensuite la réalisation de deux idées originales, du moins pour les TO7, c'est-à-dire la création de sous-programmes à variables locales (ce qui supprime un des plus graves défauts du langage BASIC classique), et surtout l'utilisation de "SPRITES" (objets programmables mobiles), qui comblera le principal manque du BASIC TO7 en matière de graphismes.

Signalons enfin que notre but ici n'est pas de donner toutes les modifications possibles, ou de recréer par exemple le BASIC disque ! les modifications présentées nous ont semblée à la fois les plus utiles et les plus "pédagogiques" ; il appartiendra au lecteur d'en imaginer et d'en réaliser d'autres, chose parfaitement possible à partir du moment où l'on connaît bien son interpréteur...

1

Créer et utiliser un BASIC français

On a vu qu'il suffit de modifier le contenu des 6 octets situés en \$6201 et \$6206 pour pouvoir utiliser respectivement des noms d'instructions et de fonctions différents de ceux d'origine.

On peut bien sûr en profiter pour créer un vocabulaire français simplifié, qui facilitera beaucoup l'apprentissage de la programmation, par exemple à de jeunes enfants: EFFACE ou DEMANDE seront compris et mémorisés plus facilement que CLS ou INPUT, et DEFDBL ou HEX\$ ne sont sans doute pas indispensables pour un débutant !

1. Création du nouveau vocabulaire

Nous proposons le programme suivant, créant un sous-ensemble français déjà très complet des instructions et fonctions BASIC les plus courantes.

```

10 REM      BASIC francais
20 CLEAR,&H7D00
25 '
30 '.....Table des instructions.....
40 C0=&H80:C1=&HD5:ADR=&H7D20:G0 SUB 200
45 '
50 '.....Table des fonctions.....
60 C0=&H80:C1=&HA6:ADR=&H7F00:G0 SUB 200
70 POKE ADR,0
95 '
100 'Initialisations(voir Plus loin)
150 END
195 '
200 '.....Creation d'une table.....
210 FOR NUM=C0 TO C1
220 READ A$
230 FOR K=1 TO LEN(A$)-1
240 CODE=ASC(MID$(A$,K))
250 POKE ADR,CODE:ADR=ADR+1:NEXT
295 '
300 'Derniere lettre ou mots inutilises
310 CODE=ASC(RIGHT$(A$,1))+&H80:K=1
320 IF VAL(A$)<>0 THEN K=VAL(A$):CODE=&HFF:NUM=NUM+K-1
330 FOR L=1 TO K
340 POKE ADR,CODE:ADR=ADR+1:NEXT
350 NEXT NUM:RETURN
595 '
600 'Vocabulaire...
610 DATA FIN,REPETE,ENCORE,DONNEES,TABLEAU,CHERCHE,FAIS
,VA,DEMARRE,"SI ",RESTAURE,REVIENS,REMARQUE,',ARRETE,SI
NON,6,SELON,4
620 DATA ENLEVE,CURSEUR,EFFACE,1,DESSINE,2,EXECUTE,SONN
E,COULEUR,TRAIT,BOITE,3,GARNIS,ECRIS,CONTINUE,LISTE,RES
ERVE,2,NETTOIE,COPIE,CHARGE,3,REGARDE,STYLO,JOUE
630 DATA COLONNE(,JUSQUA,FAIRE,7,ALORS,1,PAR,+,-,*,/,^,
ET,OU,5,>,<
640 DATA SIGNE,1,ABS,LIBRE,RACINE,6,LONGUEUR,1,VAL,ASC,
CARACT$,1,ENTIER,5,MANETTE,BOUTON,1,GAUCHE$,DROITE$,EXT
RAIT$,2,HASARD,CLAVIER$,DEMANDE,1,POINT,ECRAN,2

```

Nous donnons, ci-après, un petit programme donnant la table de traduction; les principales instructions sont (un terme placé entre crochets est facultatif):

- REPETE variable = X JUSQU'À Y [PAR Z]
-
-
-
- ENCORE [variable] (FOR...TO...STEP; NEXT)
- SI expression ALORS...SINON...(IF...THEN...ELSE...)
- VA FAIRE numéro
-
-
-
-
- REVIENS (GOSUB;RETURN)
- VA JUSQUA numéro (GOTO)

REMARQUE 1 : Les chiffres dans les DATA représentent le nombre de mots consécutifs que l'on supprime ; ils permettent d'obtenir les codes corrects pour nos mots-clés.

REMARQUE 2 : Il est bien sûr possible de compléter le vocabulaire donné ici, ou d'utiliser des mots différents ; il suffira pour cela de modifier les DATA.

Il faudra toutefois alors prendre garde à ne jamais réemployer comme premières lettres d'un mot un mot déjà défini, de code inférieur.

Par exemple, si l'on emploie LIS pour traduire READ, le mot LISTE (LIST) sera compris comme LIS TE (READ TE), d'où une erreur.

Ceci explique aussi que l'on ajoute dans la table un espace après SI (à cause de SINON), qu'il sera donc obligatoire de toujours écrire.

II. Initialisation-utilisation

1. Sous-programmes d'initialisation

Pour rendre notre vocabulaire opérationnel, il suffit de changer les adresses des tables, ce qui sera fait automatiquement par la routine suivante :

INITFR	CC,7D20	LDD	# \$7D20	
	FD,6202	STD	\$6202	Table des instruct.
	CC,7F00	LDD	# \$7F00	
	FD,6207	STD	\$6207	Table des fonctions
	39	RTS		

Un simple EXEC INITFR rendra donc votre vocabulaire opérationnel.

On peut aussi souhaiter revenir à un moment donné au vocabulaire normal; ceci sera fait par la routine suivante:

INITANG	CC,0092	LDD	# \$092
	FD,6202	STD	\$6202
	CC,01CF	LDD	# \$1CF
	FD,6207	STD	\$6207
	39	RTS	

Pour implanter ces deux routines (respectivement en \$7D01 et \$7D11, où l'on a réservé 31 octets), on reprendra le programme BASIC précédent, dans lequel on ajoutera (voir 1^{ère} partie):

```

100 '.....S.P. d'initialisation.....
110 FOR K=&H7D01 TO &H7D1D
120 READ A$:POKE K,VAL("&H"+A$):NEXT
895 '
900 'Initialisations...
910 DATA CC,7D,20,FD,62,2,CC,7F,0,FD,62,7,39,12,12,12
920 DATA CC,0,92,FD,62,2,CC,1,CF,FD,62,7,39

```

REMARQUE: Les pointeurs \$6202 et \$6207 peuvent bien sûr aussi être initialisés par des POKE.

2. Utilisation

Après avoir fait exécuter le programme BASIC précédent complété par les instructions ci-dessus, on enregistrera sur cassette les octets de \$7D01 à \$7FFF; on fera pour cela, *une fois pour toutes*:

```
SAVEM "FRANÇAIS", &H7D01, &H7FFF, 0
```


Lorsqu'on voudra travailler en français, il suffira donc de lire la cassette (6 secondes de lecture seulement) par :

```
LOADM "FRANÇAIS"
```

on fera ensuite, par exemple en mode direct :

```
CLEAR, &H7D00: EXEC &H7D01
```

on disposera alors de tous les mots-clés français pour écrire son propre programme.

On peut noter qu'un programme écrit normalement sera traduit automatiquement en français par ce procédé, lorsqu'on demandera LISTE !

De même, un programme écrit en BASIC français sera traduit automatiquement en BASIC normal par EXECUTE &H7D11:LIST.

A titre d'exemple, après avoir fait exécuter le programme BASIC précédent et fait EXEC &H7D01, vous aurez la surprise de voir ce programme listé par la commande LISTE sous la forme suivante :

```
10 REMARQUE      BASIC francais
20 RESERVE,&H7D00
25 '
30 '.....Table des instructions.....
40 C0=&H80:C1=&HD5:ADR=&H7D20:VA FAIRE 200
45 '
50 '.....Table des fonctions.....
60 C0=&H80:C1=&HA6:ADR=&H7F00:VA FAIRE 200
70 GARNIS ADR,0
95 '
100 '.....S.P. d'initialisation.....
110 REPETE K=&H7D01 JUSQUA &H7D1D
120 CHERCHE A$:GARNIS K,VAL("&H"+A$):ENCORE
150 FIN
195 '
200 '.....Creation d'une table.....
210 REPETE NUM=C0 JUSQUA C1
220 CHERCHE A$
230 REPETE K=1 JUSQUA LONGUEUR(A$)-1
240 CODE=ASC(EXTRAITS(A$,K))
250 GARNIS ADR,CODE:ADR=ADR+1:ENCORE
295 '
300 'Derniere lettre ou mots inutilises
310 CODE=ASC(DROITE$(A$,1))+&H80:K=1
320 SI VAL(A$)<>0 ALORS K=VAL(A$):CODE=&HFF:NUM=NUM+K-1
325 'etc...
```

III. Table de traduction

Elle est obtenue par un petit programme BASIC, qui liste les mots français et anglais correspondants :

FIN	:END	REPETE	:FOR
ENCORE	:NEXT	DONNEES	:DATA
TABEAU	:DIM	CHERCHE	:READ
FAIS	:LET	VA	:GO
DEMARRE	:RUN	SI	:IF
RESTAURE	:RESTORE	REVIENS	:RETURN
REMARQUE	:REM	'	:'
ARRETE	:STOP	SINON	:ELSE
SELON	:ON	ENLEVE	:DELETE
CURSEUR	:LOCATE	EFFACE	:CLS
DESSINE	:PSET	EXECUTE	:EXEC
SONNE	:BEEP	COULEUR	:COLOR
TRAIT	:LINE	BOITE	:BOX
GARNIS	:POKE	ECRIS	:PRINT
CONTINUE	:CONT	LISTE	:LIST
RESERVE	:CLEAR	NETTOIE	:NEW
COPIE	:SAVE	CHARGE	:LOAD
REGARDE	:INPEN	STYLO	:PEN
JOUE	:PLAY	COLONNE	:TAB
JUSQUA	:TO	FAIRE	:SUB
ALORS	:THEN	PAR	:STEP
+	:+	-	:-
*	:*	/	:/
^	:^	ET	:AND
OU	:OR	>	:>
=	:=	<	:<
SIGNE	:SGN	ABS	:ABS
LIBRE	:FRE	RACINE	:SQR
LONGUEUR	:LEN	VAL	:VAL
ASC	:ASC	CARACT\$:CHR\$
ENTIER	:CINT	MANETTE	:STICK
BOUTON	:STRIG	GAUCHE\$:LEFT\$
DROITE\$:RIGHT\$	EXTRAIT\$:MID\$
HASARD	:RND	CLAVIER\$:INKEY\$
DEMANDE	:INPUT	POINT	:POINT
ECRAN	:SCREEN		

2

Créer de nouvelles fonctions

Nous présentons ici une méthode générale permettant de créer à partir du BASIC de nouvelles fonctions, que l'on pourra ensuite utiliser exactement comme les fonctions disponibles au départ.

1. Principe de la méthode

On pourrait appliquer ici la méthode que nous décrirons en détail au chapitre 4 et qui nous servira à définir de nouvelles instructions.

Nous avons vu en effet, lors de l'étude du traitement des fonctions, qu'un code supérieur à &HFFA6 provoque le branchement à l'adresse située en &6213 (normalement \$7F3, d'où un "SN Error"); si l'on ajoute donc de nouveaux noms de fonctions à la fin de la table des mots-clés, il suffit de placer la valeur α en \$6213 pour que l'interpréteur se déroute en α en rencontrant une nouvelle fonction; le traitement serait donc écrit en α .

Cette méthode est la meilleure lorsqu'on doit créer un nombre relativement grand (par exemple 5 ou 6) de fonctions nouvelles.

Dans le cas contraire, il est beaucoup plus simple d'utiliser le fait que le BASIC décode comme on l'a vu le mot-clé FN (correspondant normalement à des "fonctions utilisateurs"), traité en \$623C (le "traitement" consistant en un JMP \$7F3 qui provoque un "SN Error").

On placera donc simplement en \$623C une instruction JMP α qui provoquera le déroutement de l'interpréteur à chaque rencontre d'un nom de fonction commençant par FN ; les traitements correspondants seront bien entendu implantés en α .

REMARQUE: Nous verrons au chapitre IV que l'instruction placée en \$623C n'est utilisée que par FN ; nous ne perturberons donc pas le fonctionnement de l'interpréteur en la modifiant.

II. Exemples d'applications

1. Conversions radiants-degrés

Le principe décrit ci-dessus va être appliqué à la création des deux fonctions suivantes, absentes du BASIC TO7 :

- FNR(X) : renvoie la valeur réelle exprimée en radiants correspondant à la valeur de l'argument X exprimé en degrés ;
- FND(X) : fonction inverse de la précédente ; elle convertit la valeur de l'argument X exprimé en radiants en une valeur exprimée en degrés.

Il est évident que les routines données ci-après utilisent au maximum les routines de l'interpréteur, par exemple pour vérifier la syntaxe et calculer l'argument (\$7DE), puis le convertir en réel (\$252B=CSNG) et l'envoyer dans le second accumulateur flottant situé en \$6163 (\$1C81).

Nous utilisons aussi bien sûr, les routines de traitement des opérateurs * et /, situées respectivement en \$25B6 et \$920 ; rappelons que notre étude de l'interpréteur nous a montré que ces routines doivent être appelées pour des arguments réels avec le premier opérande placé dans le second accumulateur flottant et le second opérande dans l'accumulateur flottant "normal" situé en \$6155 (on peut d'ailleurs remarquer qu'il faut faire l'inverse pour des arguments entiers : voir \$87A).

Le traitement des deux fonctions est donc le suivant :

<u>BA40</u>	9D	B2	JSR	\$B2	R ou D
<u>BA42</u>	34	02	PSHS	A	
<u>BA44</u>	9D	B2	JSR	\$B2	Pointe sur "("
<u>BA46</u>	BD	07DE	JSR	\$07DE	Argument → AC flottant
<u>BA49</u>	BD	252B	JSR	\$252B	Conversion en réel
<u>BA4C</u>	BD	1C81	JSR	\$1C81	AC flottant → \$6163
<u>BA4F</u>	CC	788E	LDD	#\$788E	Début $\pi/180$
<u>BA52</u>	DD	55	STD	\$55	
<u>BA54</u>	F	5D	CLR	\$5D	Signe +
<u>BA56</u>	CC	FA35	LDD	#\$FA35	Fin $\pi/180$
<u>BA59</u>	DD	57	STD	\$57	
<u>BA5B</u>	35	02	PULS	A	R ou D ?
<u>BA5D</u>	81	52	CMPA	#\$52	Code de "R"
<u>BA5F</u>	26	03	BNE	\$BA64	D ?
<u>BA61</u>	7E	25B6	JMP	\$25B6	Traite * et RTS
<u>BA64</u>	81	44	CMPA	#\$44	Code de "D"
<u>BA66</u>	26	03	BNE	\$BA6B	Ni R, ni D
<u>BA68</u>	7E	0920	JMP	\$0920	Traite / et RTS
<u>BA6B</u>	7E	07F3	JMP	\$07F3	SN Error

REMARQUE: Les 2^e et 3^e colonnes représentent le code objet, à implanter en mémoire; il a été bien sûr facilement établi "à la main", à partir des tableaux que l'on trouvera en annexe, donnant les codes des instructions et les parties adresses de certaines opérations particulières (adressages relatifs ou indexés, PSHS, TFR, etc...).

2. D'autres idées

Toujours comme fonctions de calcul, on pourrait écrire facilement une fonction "Arc tangente" (formule donnée dans le manuel de référence BASIC), ou une fonction "affine" rapide : FNA (a,b,X) calculerait automatiquement $aX+b$, environ deux fois plus rapidement que par $a * X + b$: voir pour cela INC, au chapitre V ; SQR étant très lent, on pourrait encore créer une fonction "racine carrée entière", par exemple à partir de l'algorithme du chapitre 3.

On pourrait aussi créer une fonction (souvent appelée DEEK) retournant la valeur contenue dans les deux octets situés à l'adresse spécifiée par l'argument, ou retournant la plus grande (ou la plus petite) valeur d'un couple d'arguments, ou encore recherchant l'occurrence d'une valeur dans un

tableau numérique ou alphanumérique: FNO (Nom de tableau, valeur) retournerait le rang où se situe la valeur, ou — 1 par exemple.

Toutes ces fonctions peuvent être bien entendu écrites en BASIC, par exemple par `256 * PEEK(I) + PEEK(I+1)` pour DEEK; les écrire une fois pour toutes en langage machine et les obtenir par FNx est évidemment ensuite beaucoup plus simple et beaucoup plus rapide à l'exécution !

III. Création des nouvelles fonctions

On pourrait bien sûr implanter la routine précédente comme on l'a fait au chapitre I pour les sous-programmes d'initialisation (lignes 110 et 120); l'adresse d'implantation serait alors placée en \$623D par deux POKE.

Nous préférons donner le programme suivant, qui permet de ranger à des adresses quelconques de la mémoire un nombre lui-même quelconque de routines écrites en langage machine, et cela comme nous allons le voir de la manière la plus rapide, la plus souple et la plus "lisible" possible.

Ce programme sera utilisé pour toutes les applications à suivre (nous le complèterons au chapitre IV pour créer de nouveaux mots-clés), en respectant à chaque fois la procédure suivante :

- chaque routine sera écrite en DATA, à partir de la ligne 2000; les valeurs hexadécimales du code objet, toujours écrites sans &H pour une commodité maximale, seront précédées de l'adresse d'implantation de la routine, qui devra elle être écrite avec &H (évitant ainsi toute confusion ou oubli);
- pour faciliter la relecture et les corrections éventuelles, les codes hexadécimaux pourront être regroupés par 2 ou 3 par exemple; on pourra donc traduire:
 - `JMP $25B6` par `73,25B6` ou `7325B6`
 - l'adresse `$7F3` indifféremment par `7F3` ou `07F3`
 - la valeur `#$0001` (2 octets) par `001` ou `0001` (pas `01` bien sûr, qui n'occuperait qu'un octet).
- le premier DATA de chaque routine commencera par un mot (guillemets inutiles) identifiant la routine, qui précèdera donc l'adresse d'implantation et les codes;
- la liste des codes de chaque routine sera terminée par une phrase commençant par FIN ("FIN fonctions" par exemple).

Exemple: Nous écrivons donc ici:

2000 DATA Fonctions, &HBA40,9D,B2,34,2,9D,B2
,BD,7DE,....,7E,7F3,FIN Fonctions

(l'adresse BA40 sera bien entendu remplacée par 7A40 si l'on ne dispose pas de l'extension mémoire, ou par \$DA40 pour le TO7-70).

- la dernière routine sera suivie d'un DATA contenant une phrase commençant par FIN ("FIN des routines", par exemple), qui arrêtera le programme;
- enfin, ces nouvelles routines doivent toujours être activées par un sous-programme d'initialisation, dont le DATA sera placé en premier (en 1700) et écrit exactement de la même manière.

Dans le cas de nos fonctions FNx, le sous-programme est simplement:

INITF	CC,BA40	LDD	#\$adr.	Début routine
	FD,623D	STD	\$623D	
	39	RTS		

Le programme complet est donc le suivant:

```
1000 REM   Creation de routines
1010 CLEAR:&HBA00      '...ou 7A00(TO7) ou DA00(TO7-70)
1020 DEB=256*PEEK(&H612A)+PEEK(&H612B)+2:FIN=256*PEEK(&H65AC)+PEEK(&H65AD)+1:ADR0=DEB
1030 READ NOM$:IF LEFT$(NOM$,3)="FIN" GOTO1300
1040 READ ADR   'Adresse d'implantation
1050 IF ADR<ADR0 OR ADR>FIN THEN PRINT"ERREUR D'ADRESSE
...":END
1060 PRINT "ROUTINE ";NOM$
1070 PRINT" Debut:";HEX$(ADR),
1095 '
1100 'Lecture et rangement d'une routine
1110 S=0:FOR I=ADR TO FIN
1120 READ A$:L=LEN(A$):IF LEFT$(A$,3)="FIN" GOTO 1200
1130 IFL<=2 GOTO1160
1140 K=2-L MOD 2:V=VAL("&H"+LEFT$(A$,K)):POKE I,V:S=S+V:I=I+1
1150 L=L-K:A$=RIGHT$(A$,L):GOTO1130
1160 V=VAL("&H"+A$):S=S+V:POKE I,V:NEXT
1200 PRINT"   Fin:";HEX$(I-1),
1210 PRINT" Somme";S
1220 PRINT:ADR0=I
```

```

1230 GOTO1030          'Routine suivante
1300 '
1595 '
1600 '.....Terminer.....
1610 PRINT " Implantation terminée..." :PRINT " Enregistr
er de ";HEX$(DEB);" a ";HEX$(ADR0-1);" (Par SAVEM)":END

```

Dans le cas des fonctions, les DATA sont les suivants:

```

1690 '
1695 '.....Initialisation.....
1700 DATA Initialisation,&HBA01,CC,BA40,FD,623D,39,FIN
1990 '
1995 'Routines
2000 DATA Fonctions,&HBA40,9D,B2,34,2,9D,B2,BD,7DE,BD,2
52B,BD,1C81,CC,7B8E,DD,55,F,5D,CC,FA35,DD,57,35,2,81,52
,26,3,7E,25B6,81,44,26,3,7E,920,7E,7F3,FIN
2950 DATA FIN des routines

```

REMARQUE: On constatera que le programme vérifie systématiquement toutes les adresses, évitant ainsi tout recouvrement ou écriture dans une zone protégée ou inexistante (celà grâce aux adresses contenues en \$612A et \$65F5: voir annexes) On notera aussi que le programme calcule une somme de contrôle permettant de vérifier les DATA; on doit avoir ici 4888 pour la routine "Fonctions".

IV. Utilisation du programme de création

1. Quelques conseils

A chaque utilisation du programme précédent pour créer une routine quelconque, il faudra toujours sauver le programme avec ses DATA *avant* toute exécution du sous-programme d'initialisation; une erreur dans celui-ci ou dans la routine "planterait" en effet la machine, obligeant dans la plupart des cas à couper le courant...

On pourra alors écrire en début de programme (lignes 1 à 999) les instructions BASIC permettant de tester la routine, après avoir fait exécuter le sous-programme d'initialisation (par EXEC &HBA01 ici).

Rappelons que pour la mise au point d'une nouvelle routine, il est bien sûr possible d'utiliser le désassembleur pour vérifier que les codes sont corrects ; on pourra aussi par exemple intercaler au milieu un ou plusieurs appels de la routine \$1ED1, qui écrit (en décimal) la valeur (convertible en hexadécimal par HEX\$), placée dans l'accumulateur D du 6809 (et place la valeur 3 dans \$6105) ; on pourra ainsi facilement vérifier le contenu des registres (par TFR registre, D) ou de certaines mémoires.

Rappelons aussi qu'un JMP \$2AED provoquera le retour au BASIC, permettant par exemple de vérifier la pile (adresse du sommet : PEEK (&H618C) * 256 + PEEK (&H618D)).

On peut aussi utiliser l'instruction SWI pour créer des arrêts sur adresse : voir 1^{re} partie.

Enfin, on fera bien entendu ré-exécuter le programme de création après toute modification dans les DATA, afin de modifier effectivement la routine en mémoire !

2. Utilisation des nouvelles routines

Le programme de création signale à la fin de l'exécution la zone de mémoire à enregistrer en binaire, ce qu'on fera *une fois pour toutes* par :

```
SAVEM "Nom",&HAdresse 1,&HAdresse 2,0
```

Dans notre cas, "Adresse 1" (\$BA01 ici) est l'adresse du sous-programme d'initialisation ; pour pouvoir utiliser les nouvelles fonctions dans un programme BASIC, il suffit donc de faire lire le programme binaire "Nom" par : LOADM "Nom" (ce qui ne demande que quelques secondes), puis de faire exécuter l'instruction :

```
CLEAR,&HAdresse 1 - 1: EXEC &HAdresse 1
```

Ceci activera les nouvelles fonctions.

3. Exemple: FNR et FND

Avec notre routine, on pourra écrire par exemple :

```
100 X=180:PRINT X;"degres =" ;FNR(X);"radians"
110 X=73:PRINT " VERIF.:";X;"degres =" ;FND(FNR(X));"deg
res"
120 END
```

On obtient bien sûr :

```
180 degres = 3.14159 radians
VERIF.: 73 degres = 73 degres
```

REMARQUE: Les calculs sont en fait effectués avec la valeur $\pi = 3.1415927$, exacte à mieux que 10^{-7} près.

3

Boucle WHILE... WEND

Les deux instructions WHILE et WEND permettent de boucler sur une séquence d'instructions *tant qu'*une certaine condition, écrite après le WHILE, est réalisée : lorsqu'elle ne l'est plus, il y a branchement à l'instruction qui suit le WEND.

Ceci permet donc de réaliser des boucles différentes de celles contrôlées par FOR et NEXT : leur principal intérêt est de permettre une bonne structuration des programmes, cette notion étant comme on le sait fondamentale en programmation ; elle facilite en effet énormément l'écriture, la mise au point et les modifications ultérieures d'un programme.

Signalons d'ailleurs que la possibilité d'écrire des sous-programmes à variables locales, que nous verrons au chapitre VI, constitue un autre élément très important de la programmation structurée.

1. Principe de création d'une boucle WHILE

La routine correspondante, dont le principe est proche de celui d'une boucle FOR, utilise la routine \$16AC qui trouve dans tous les cas le WEND associé au WHILE (voir la 2^e partie).

Lors du traitement de WHILE, on empile donc successivement :

- l'adresse du caractère qui suit le WHILE,
- le numéro de la ligne du WHILE (ces deux informations permettant de se repositionner au retour du WEND),
- l'adresse du caractère qui suit le WEND associé,
- la valeur &HAF, c'est-à-dire le code de WHILE (ceci rend notre routine compatible avec la routine \$2F3, appelée par FOR et GOSUB).

On appelle ensuite la routine de traitement du WEND, qui teste la condition ; la boucle ne sera donc exécutée que si celle-ci est vraie.

Le traitement de WEND consiste essentiellement à tester la condition de contrôle de la boucle :

- si elle est vraie, on se repositionne au début de la boucle et on fait un JMP \$2AED qui provoque l'exécution des instructions correspondantes (voir 2^e partie),
- si elle est fausse, on reste positionné après le WEND, on dépile les 7 octets empilés par le WHILE et on fait encore JMP \$2AED.

Enfin, les deux routines correspondant à WHILE et WEND seront activées par un sous-programme d'initialisation modifiant \$6233 (WHILE) et \$6236 (WEND), contenant initialement une instruction JMP \$7F3 (SN Error) correspondant au "traitement" de WHILE et WEND.

Signalons que nos routines permettent certaines "libertés" d'une boucle FOR, c'est-à-dire les boucles imbriquées et l'écriture éventuelle du WEND après un THEN ou un ELSE.

Toutefois, en cas de boucles imbriquées, les sorties anormales (qui ne passent pas par le WEND, à cause d'un IF ou d'un GOTO) sont volontairement interdites (on aura un WE Error), ceci d'une part parce que la condition de sortie correspondante peut être en principe ajoutée à la condition de contrôle et d'autre part parce que ceci n'a pas de raison d'être dans des programmes structurés !

Pour la même raison, nous n'avons pas prévu ici de dépilement en cas de sortie anormale (alors que c'est le cas pour FOR, comme on l'a vu dans la seconde partie), qui reste toutefois autorisée pour une boucle simple.

REMARQUE: Si l'on voulait réaliser une boucle REPEAT... UNTIL condition (répéter... jusqu'à ce que condition; cette boucle est toujours décrite au moins une fois), le principe à utiliser serait pratiquement le même que celui donné ci-dessus.

II. Les routines-initialisation

Les routines correspondant à WHILE et WEND (enchaînées, comme on va le voir) seront bien sûr implantées par le programme de création précédent.

Le sous-programme d'initialisation est ici le suivant:

INITW	CC,BA70	LDD	# \$adr.	Adresse routine
	FD,6234	STD	\$6234	Adr. trait. WHILE
	C3,01E	ADDD	# \$001E	30 octets pour WHILE
	FD,6237	STD	\$6237	Adr. trait. WEND
	39	RTS		

Si l'on désire donc par exemple implanter simultanément les fonctions précédentes (en \$BA40 toujours) et le traitement de WHILE en \$BA70, on enlèvera ",39,FIN" de 1700 et on ajoutera:

```
1710 DATA CC,BA70,FD,6234,C3,01E,FD,6237,39,FIN
```

La routine proprement dite sera ensuite écrite en 2100 et 2110, sous la forme:

```
2100 DATA WHILE-WEND,&HBA70, codes, FIN
```

Le listing est le suivant:

BA70	32	62	LEAS	2,S	réinitialise pile
BA72	06	04	LDB	##04	2 x 4 octets à empiler
BA74	BD	0336	JSR	\$0336	Débordement ?
BA77	DE	B9	LDU	\$B9	Caract. après WHILE
BA79	9E	2C	LDX	\$2C	N° ligne du WHILE
BA7B	34	50	PSHS	X,U	Empilement
BA7D	BD	16AC	JSR	\$16AC	Trouve le bon WEND

BAB0	9E	78	LDX	\$78	N° ligne du WEND
BAB2	9F	2C	STX	\$2C	
BAB4	9D	B2	JSR	\$B2	Caract. après WEND
BAB6	DE	B9	LDU	\$B9	Adresse caractère
BAB8	86	AF	LDA	##AF	Code de WHILE
BABA	34	42	PSHS	A, U	Empilement
BABC	8D	00	BSR	##BA8E	WEND
BABE	32	62	LEAS	2, S	Enlève adr. retour
BAB0	AE	61	LDX	1, S	Adr.car. après WEND → X
BAB2	9C	B9	CMPX	\$B9	Est-ce le même ?
BAB4	27	05	BEQ	##BA9B	Oui
BAB6	06	19	LDB	##19	Non → erreur
BAB8	7E	0353	JMP	\$0353	WE Error
BAB8	AE	65	LDX	5, S	Caract. après WHILE
BABD	9F	B9	STX	\$B9	On se repositionne
BABF	BD	081A	JSR	\$081A	Calcul condition
BAA2	ED	1CC8	JSR	\$1CC8	Faux ⇔ Z=1 (Z=0 sinon)
BAA5	27	07	BEQ	##BAAE	Faux → terminé
BAA7	AE	63	LDX	3, S	N° ligne du WHILE
BAA9	9F	2C	STX	\$2C	
BAB8	7E	2AED	JMP	\$2AED	Exécution boucle
BAAE	AE	61	LDX	1, S	Caract. après WEND
BAB0	32	67	LEAS	7, S	Dépilement
BAB2	7E	1669	JMP	\$1669	STX \$B9, JMP \$2AED

REMARQUE: On doit obtenir ici 7613 comme somme de contrôle.

III. Exemple d'utilisation

On fera bien sûr exécuter le programme de création et on enregistrera le résultat une fois pour toutes sur cassette (par SAVEM "Nom", &HBA01, &HBAB4,0).

Lorsqu'on voudra utiliser une boucle WHILE, il suffira donc de faire lire notre routine (par LOADM "Nom").

On pourra alors par exemple écrire le programme suivant:

```
10 CLEAR,&HBA00:EXEC&HBA01
20 'Calcul de racine carree entiere
30 X=15913
```

```

40 R=10:WHILE R<X:R=R+10:WEND
50 'On a atteint ou depasse la racine
60 WHILE R>X:R=R-1:WEND
70 PRINT" Racine de";X;"=";R

```

On obtient bien sûr 126, ce programme ayant pour seule ambition de montrer la logique (et la clarté !) d'une boucle WHILE.

On notera quand même que cet algorithme élémentaire traduit en langage machine calculerait des racines entières beaucoup plus rapidement que SQR ; on pourrait donc l'utiliser par exemple pour créer une instruction de tracé de cercle.

4

Création et utilisation de nouvelles instructions

Nous nous intéressons ici à la création d'instructions nouvelles, non décodées par le BASIC comme l'étaient FN, WHILE et WEND.

Les routines correspondantes seront étudiées dans les chapitres suivants.

1. Principe de la méthode

La méthode la plus simple consiste à ajouter les nouveaux mots à la fin de la table des fonctions (qui contient déjà les *instructions* MID\$, INPUT et SCREEN); celle-ci sera donc recopiée dans la RAM, et les nouveaux mots ajoutés ensuite.

Ces mots écrits dans un programme seront alors automatiquement codés par &HFFA7, FFA8, etc...; lors de l'exécution du programme, la routine \$2B25 de traitement des instructions branchera donc en \$6273 (voir 2^e partie), où l'on écrira une instruction JMP adresse.

En "adresse", il suffira finalement d'écrire une routine de branchement vers les traitements des nouvelles instructions; on utilisera pour cela une table des nouvelles adresses.

REMARQUE 1: Il faut être sûr de ne pas modifier le fonctionnement du BASIC en changeant les octets \$6273 à \$6275.

On reprend donc le programme 1 de la 2^e partie, en modifiant la ligne 40 pour obtenir toutes les valeurs comprises entre &H6273 et &H6275 (...IF PEEK(I+1) ≥ &H72 AND PEEK(I+1) ≤ &H76 THEN...).

On obtient seulement \$2B62, contenant JMP \$6273; \$6273 contient lui-même RTS, suivi donc de deux octets inemployés. Nous pouvons donc modifier ces 3 octets.

REMARQUE 2: Le même programme cherchant toutes les valeurs de &H626D à &H62A8 (points de contrôle de diverses routines: voir annexes) permet de trouver de même toutes les utilisations des octets correspondants (contenant 20 fois les trois valeurs &H39 (RTS), 2E et 74).

On constate que les seules instructions utilisant une de ces adresses sont toutes de la forme:

JSR (ou JMP) \$626D+3x, avec x=0,1..., 19

Elles pointent donc toutes sur un RTS, les deux octets suivants étant inutilisés; de plus, chacune de ces adresses n'est utilisée qu'en un seul endroit de l'interpréteur.

Ceci nous autorisera à mettre en œuvre les modifications de la quatrième partie de notre ouvrage.

Signalons enfin que le même contrôle appliqué aux octets de \$6233 à 623E (WHILE, WEND, DEFFN (voir DEF, en \$1801) et FN) aboutit à la même conclusion, ce qui justifie à posteriori les interventions des chapitres II et III.

II. Mise en œuvre de la méthode

La méthode que nous venons de décrire sera bien sûr mise en œuvre grâce au programme de création, complété comme nous allons l'indiquer; la nouvelle table des noms de fonctions et la table des adresses des nouvelles instructions seront alors créées automatiquement, et cela encore une fois de la manière la plus souple possible.

1. Branchement au traitement des nouvelles instructions

Lors de l'exécution d'un programme utilisant les nouvelles instructions, le branchement aux différentes routines s'effectue grâce à un petit sous-programme, placé juste après celui d'initialisation; ce sous-programme utilise une table des adresses des nouvelles instructions (placée en ADRA), exactement comme le fait la routine \$2B25 (d'où l'utilisation de \$2B38: voir 2^e partie).

Cette table des nouvelles adresses sera suivie elle-même de la nouvelle table des noms de fonctions, placée par exemple ici 24 octets plus loin, ce qui permet la création de 12 instructions nouvelles.

Les sous-programme de branchement est donc le suivant :

BRAN	8E,BF01	LDX	#ADRA	Adr. table des adr.
	80,A7	SUBA	# \$A7	x ^e mot \Rightarrow (x-1) \rightarrow A
	2A,3	BPL	03	Nouveau mot
	7E,7F3	JMP	\$7F3	SN Error
NOUV	73,2B38	JMP	\$2B38	Branche au trait.

REMARQUE: La table des adresses est placée automatiquement (voir III ci-après) en \$7F01 pour le TO7 de base, en \$BF01 pour le TO7 avec extension mémoire et en \$DF01 pour le TO7-70.

Selon le cas, on devra donc remplacer ADRA par ces valeurs.

2. Activation des nouvelles instructions

Le sous-programme d'initialisation doit modifier l'adresse contenue en \$6207 (début de la table des noms de fonctions), la valeur contenue en \$6206 (nombre de mots, placé par le programme de création en ADRA-1), et placer une instruction JMP BRAN en \$6273, le sous-programme BRAN précédent étant implanté juste après.

On écrira donc :

INITNV	CC,BF19	LDD	#ADRN	Adr. table des noms
	FD,6207	STD	\$6207	
	B6,BF00	LDA	ADRA-1	Nombre de mots
	B7,6206	STA	\$6206	
	86,7E	LDA	# \$7E	Code de JMP étendu
	B7,6273	STA	\$6273	

E	1F,50 C3,007 FD,6274 39	TFR ADDD STD RTS	PC,D # \$0007 \$6274	Adresse E → D (Adresse RTS)+1 → D
---	----------------------------------	---------------------------	----------------------------	--------------------------------------

REMARQUE 1 : On observera l'utilisation du compteur de programme PC du 6809, permettant de déplacer les sous-programmes d'initialisation et de branchement sans avoir à les modifier.

REMARQUE 2 : Pour la même raison que précédemment, on écrira obligatoirement les adresses \$7F19 et \$7F00 pour le TO7 de base, et \$DF19 et \$DF00 pour le TO7-70 (au lieu de \$BF19 et \$BF00).

3. Implantation en mémoire

Finalement, si l'on désire implanter simultanément les nouvelles fonctions, les boucles WHILE et certaines nouvelles instructions, dont la table des adresses sera placée en cas d'extension mémoire en \$BF00 (voir ci-après), on enlèvera ",39,FIN" de la ligne 1710 et l'on ajoutera :

```
1720 DATA CC,BF19,FD,6207,B6,BF00,B7,6206,86,7E,B7,6273
,1F,50,C3,007,FD,6274,39
1730 DATA 8E,BF01,80,A7,2A,3,7E,7F3,7E,2B38,FIN Initial
isations
```

On peut aussi bien sûr n'implanter que les nouvelles instructions, puisqu'on peut déplacer le sous-programme d'initialisation sans modifications ; on écrira alors simplement :

```
1700 DATA Initialisation,&HBA01,CC,BF19,etc...
```

III. Création des nouvelles tables

Les mots-clés correspondants aux nouvelles instructions à créer seront placés en DATA, à la fin du programme (ligne 2990), chacun étant suivi de

l'adresse hexadécimale (précédée de &H) où se situera la routine de traitement correspondante.

Ces adresses seront celles de branchement et elles devront bien sûr être les mêmes que celles écrites au début des routines proprement dites (adresses d'implantation).

Toutefois, il est aussi possible de créer des instructions "non exécutables" (par exemple ARG au chapitre VI); on écrira alors comme adresse \$7F3 (SN Error) ou \$663 (adresse de DATA: l'instruction sera ignorée).

La liste des nouvelles instructions sera toujours terminée par le mot FIN.

Les instructions BASIC suivantes, à ajouter au programme de création du chapitre 2, génèrent automatiquement la table des nouvelles adresses, implantée à la dernière page de la RAM (\$7F01 ou \$BF01); la table des noms de fonctions est implantée comme on l'a dit 24 octets plus loin.

```
1300 '.....Nouvelles instructions.....
1310 ADR=FIN-&HFE '7F01 ou BF01(T07) ou DF01(T07-70)
1320 IF ADR<ADR0 GOTO1050 ELSE ADR0=ADR
1330 NBRM=39 '39 fonctions au départ
1340 '.....Recopie table des noms.....
1350 ADR=ADR0+24:FOR I=&H1CF TO &H269
1360 POKE ADR,PEEK(I):ADR=ADR+1:NEXT
1395 '
1400 '.....On ajoute un nouveau mot.....
1410 READ A$:IF LEFT$(A$,3)="FIN" GOTO1550
1420 FOR I=1 TO LEN(A$)-1
1430 POKE ADR,ASC(MID$(A$,I)):ADR=ADR+1:NEXT
1440 POKE ADR,ASC(RIGHT$(A$,1))+&H80:ADR=ADR+1
1450 NBRM=NBRM+1 '+1 mot
1455 '
1460 'On ajoute l'adresse dans la table
1470 READ V
1480 IF V>=FIN-&HFF THEN ADR=0:GOTO1050
1490 I=INT(V/256):POKE ADR0,I
1500 I=V-I*256:POKE ADR0+1,I
1510 ADR0=ADR0+2:GOTO1400
1545 '
1550 '...Tous les nouveaux mots créés...
1560 POKE ADR,0:ADR0=ADR+1:POKE FIN-&HFF,NBRM
```

Exemple : Pour créer l'instruction INC (voir chapitre suivant), implantée \$BAB8 par exemple, il suffira d'ajouter au programme :

```
2200 DATA INC,&HBAB8,liste des codes,FIN  
2990 DATA INC,&HBAB8,FIN nouveaux mots
```

Après exécution, un simple EXEC &HBA01 nous permettra d'utiliser notre nouvelle instruction (avant l'exécution du sous-programme d'initialisation, notre nouveau mot ne serait pas reconnu).

5

Incrémentation et permutation de variables

Nous allons appliquer la méthode précédente à la création des deux instructions relativement simples INC (incrémentation rapide) et SWAP (permutation), qui pourront servir de modèle pour réaliser des instructions similaires (par exemple un calcul rapide de fonction linéaire).

Ces deux instructions contrôlent bien sûr la syntaxe et les types des variables, exactement comme les instructions "normales".

1. Instruction INC

1. Syntaxe-but

La syntaxe est la suivante :

INC variable [, expression]

Cette instruction ajoute la valeur de l'expression à la variable (qui peut bien sûr être un élément de tableau); si l'option [, **expression**] est omise, l'incrément par défaut est 1.

L'intérêt de cette instruction est d'être exactement 1,5 à 2,5 fois (selon le type de la variable et la présence ou non du deuxième argument) plus rapide que l'affectation normale, ce qui sera très précieux pour tous les programmes où la rapidité d'exécution est essentielle.

Toute instruction d'un programme de la forme $X=X+expression$ pourra donc alors être remplacée par `INC X, expression`.

2. Réalisation

La routine est donnée ci-après; on constatera bien sûr qu'elle fait encore une fois un large appel aux routines de l'interpréteur (\$A48, 1C08, 81A, etc...) dont on trouvera la description en annexe.

On remarquera aussi l'utilisation de la valeur (double ou simple précision) 0,5 située en \$2380, utilisée par les routines de conversion de type pour les arrondis; la valeur entière 1 est trouvée de même par exemple en \$F9CA (LDX #\$F9CA place donc 1 dans le bit N du registre CC, ce qui correspond bien en \$1C08 au type 2) pour le TO7.

La routine est la suivante :

BAB8	BD	0A48	JSR	\$0A48	Adr.var.→X;type→05
BAB8	9F	3F	STX	\$3F	
BABD	9D	B8	JSR	\$B8	Expression ?
BABF	26	11	BNE	\$BAD2	Calcul expression
BAC1	8E	2380	LDX	##2380	Emplacement de 0.5
BAC4	9D	CD	JSR	\$CD	Teste le type
BAC6	2A	03	BPL	\$BACB	Réel ou dble.préc.
BAC8	8E	F9CA	LDX	##F9CA	Contient 1(TO7)
BACB	BD	1C08	JSR	\$1C08	.5 ou 1→AC.flottant
BACE	C	55	INC	\$55	.5→1(réel)
BAD0	20	0E	BRA	\$BAE0	Calcul
BAD2	96	05	LDA	\$05	Type variable
BAD4	34	02	PSHS	A	Empilement
BAD6	9D	CA	JSR	\$CA	A-t'on ",,,"?
BAD8	BD	081A	JSR	\$081A	Exp.→AC flottant
BADB	35	02	PULS	A	Type
BADD	BD	2510	JSR	\$2510	Conversion exp.
BAE0	9E	3F	LDX	\$3F	Adr. variable
BAE2	BD	24FD	JSR	\$24FD	Teste le type(num.)

BAE5	29	0A	BVS	\$BAF1	Réel(simple préc.)
BAE7	24	06	BHS	\$BAEF	Double préc.
BAE9	AE	84	LDX	, X	Valeur var. (entière)
BAEB	9F	65	STX	\$65	(donne : 2 ^d AC flottant)
BAED	20	05	BRA	\$BAF4	Addition
BAEF	C	03	INC	\$03	
BAF1	BD	1ABF	JSR	\$1ABF	Valeur var. → 2 ^d AC
BAF4	BD	2590	JSR	\$2590	+
BAF7	7E	1C36	JMP	\$1C36	Résultat → variable

REMARQUE 1 : Si la variable est de type entier, l'addition est faite ici à l'envers (voir chapitre II sur les fonctions), ce qui n'a aucune importance puisque + est commutatif ! Si l'on voulait, par contre, utiliser un opérateur tel que $-$, $/$, \leq , etc..., il faudrait obligatoirement permuter (en \$BAED) les deux accumulateurs entiers (\$6157 et \$6165).

REMARQUE 2 : La valeur F9CA (adresse contenant &H0001) située en \$BAC9 devra être remplacée par EA24 pour le TO7-70.

3. Exemples

Après implantation de la routine (voir chapitre précédent) et exécution du sous-programme d'initialisation (EXEC &HBA01), on pourra écrire par exemple :

```
100 X=31.45:INC X,-2:PRINT X
110 YZ=-17:INC YZ:PRINT YZ
```

On obtient bien sûr 29.45 et -16.

II. Instruction SWAP

1. Syntaxe-but

Cette instruction, relativement classique, et particulièrement utile pour les tris, s'écrit :

SWAP variable1 , variable2

Elle permute le contenu des deux variables (qui peuvent être des éléments de tableau), avec conversion en cas de types numériques différents; les variables chaînes sont ici admises (l'erreur "TM" est bien sûr détectée et affichée si une seule des deux variables est de type chaîne).

2. Réalisation

On observera que notre routine utilise essentiellement une partie du traitement de l'affectation (\$734 et 737), ce qui est logique puisqu'il s'agit ici d'affecter la valeur de var2 à var1 et réciproquement; la permutation est effectuée à l'aide de la pile, dans laquelle on range la valeur de var1 (par STS, \$3F, JSR \$737).

La routine est la suivante :

<u>BAFC</u>	BD	0800	JSR	\$0800	Valeur var1→AC
BAFF	32	78	LEAS	-8,S	8 octets maxi
BB01	10DF	3F	STS	\$3F	Adresse rangt.valeur
BB04	96	05	LDA	\$05	Type de var1
BB06	9E	3D	LDX	\$3D	Adresse de var1
BB08	34	12	PSHS	A,X	Empilement
BB0A	BD	0737	JSR	\$0737	Valeur var1→pile S
BB0D	9D	0A	JSR	\$0A	A-t'on "," ?
BB0F	BD	0800	JSR	\$0800	Valeur var2→AC
BB12	96	05	LDA	\$05	Type de var2
BB14	97	42	STA	\$42	
BB16	35	12	PULS	A,X	Type et adr.var1
BB18	9F	3F	STX	\$3F	
BB1A	BD	0734	JSR	\$0734	Var2 convertie→var1
BB1D	1F	41	TFR	S,X	Adr.valeur var1→X
BB1F	BD	0803	JSR	\$0803	Valeur var1→AC
BB22	96	42	LDA	\$42	Type de var2
BB24	9E	3D	LDX	\$3D	Adresse de var2
BB26	9F	3F	STX	\$3F	
BB28	BD	0734	JSR	\$0734	Var1 convertie→var2
BB2B	32	68	LEAS	8,S	Restaure la pile
BB2D	39		RTS		

REMARQUE : Les variables ne sont pas créées en mémoire par cette instruction (utilisation de \$800: voir 2^e partie).

3. Utilisation-exemples

Nous rappelons ici le mode opératoire, à utiliser pour toute instruction nouvelle.

Pour implanter notre routine en \$BAFC par exemple (ou en toute autre adresse ne recouvrant pas une routine précédente, ou la dernière page de la RAM ; rappelons que ce contrôle est effectué automatiquement par le programme de création), on écrira donc :

```
2300 DATA SWAP,&HBAFC,BD,800,...,39,FIN
2990 DATA INC,&HBAB8,SWAP,&HBAFC,FIN
```

Après création, on enregistrera une fois pour toutes les octets correspondants par SAVEM, les adresses étant indiquées par le programme.

Après relecture (LOADM) et exécution de CLEAR,INIT-1 : EXEC INIT (INIT étant par exemple ici \$BA01), on pourra écrire par exemple :

```
100 X=31.87:Y=-2.19:Z%=493
110 SWAP X,Y:PRINT X,Y,Z%
120 SWAP Y,Z%:PRINT X,Y,Z%
130 A$="PAUL":B$="JEANNE"
140 IF A$>B$ THEN SWAP A$,B$
150 PRINT A$,B$:END
```

On obtiendra bien sûr :

-2.19	31.87	493
-2.19	493	32
JEANNE	PAUL	

4. Amélioration possible

La routine précédente permet de permuter deux éléments de tableaux ; on pourrait prévoir aussi la permutation directe de deux tableaux ; les noms seraient alors précédés de DIM pour indiquer à l'interpréteur qu'il faut permuter deux ensembles de valeurs (et non deux valeurs seulement).

On pourrait donc alors écrire :

```
SWAP DIM Tableau1,DIM Tableau2
```

Bien entendu, la routine vérifierait la conformité des deux tableaux, qui devraient avoir été déclarés auparavant; elle pourra être écrite en s'inspirant de la transmission des tableaux entre unités de programmes indépendantes, que nous allons voir au chapitre suivant (en particulier, utilisation de \$A48 appelée avec 1 dans \$6107).

6

Procédures variables locales

On sait que la méthode la plus efficace pour traiter un problème complexe consiste à le décomposer en plusieurs sous-problèmes indépendants les uns des autres ; chaque sous problème pourra lui-même être décomposé de la même manière en "sous-sous-problèmes", etc... jusqu'à arriver à des problèmes élémentaires que l'on pourra programmer immédiatement.

Cette méthode d'analyse "par raffinement graduel" est dite "descendante" ; elle opère donc du général vers le détail, et non en sens inverse comme on le fait hélas couramment !

Une telle approche est possible en BASIC (usage des GOSUB et non des GOTO...) même si ce langage est peu adapté pour cela ; toutefois, on est alors astreint à tenir compte à chaque niveau des informations utilisées par toutes les autres unités de programme ; il n'est donc pas possible d'écrire et de mettre au point chaque module indépendamment des autres, d'où l'impossibilité pratique d'écrire et de mettre au point des programmes longs, et la difficulté de trouver certaines erreurs conduisant à des comportements anarchiques !

Ceci est dû au fait qu'en BASIC toutes les variables d'un programme sont communes à toutes les parties de celui-ci, conduisant donc à des interac-

tions néfastes ; la seule exception est constituée pour certains BASIC par les fonctions utilisateurs FN, limitées à un simple calcul et donc sans portée réelle.

Nous donnons ici une méthode permettant de créer des unités de programme, ou *procédures*, indépendantes les unes des autres ; toutes les informations utilisées par une de nos procédures n'ont en effet de sens que pour celle-ci, c'est-à-dire qu'elles sont *locales* à la procédure.

Nous verrons bien sûr que ces procédures peuvent aussi recevoir ou échanger des informations avec le programme qui les appelle.

Elles peuvent enfin s'appeler elles-mêmes, ce qui autorise donc la vraie récursivité.

Il vaut donc la peine de faire l'effort d'écrire les seuls 375 octets nécessaires pour tout cela !...

1. Principe de la méthode – Passage des Arguments

1. Principe

On a vu dans la deuxième partie que l'interpréteur gère les informations utilisées par un programme à partir des adresses contenues en \$611E (début de la zone des variables), \$6120 (début de la zone des tableaux) et \$6122 (début DEB de la zone libre, contenant la pile).

Pour créer des variables ou des tableaux locaux à une certaine partie de programme, il suffit donc d'envoyer l'adresse DEB en \$611E et \$6120, après sauvegarde des valeurs initiales dans la pile.

Toutes les variables ou tableaux rencontrés alors seront créés automatiquement (par la routine \$A48) dans une zone de mémoire indépendante de la zone normale, et seront donc complètement différents des variables ou tableaux utilisés jusque là, et cela même dans le cas où les noms seraient les mêmes !

Bien entendu, on restaurera lors du retour au programme principal (instruction PROCEND définie ci-après) les valeurs de \$611E, \$6120 et \$6122 (cette dernière avec le contenu de \$611E, qui ne bouge pas) ; les informations utilisées par la procédure ne seront donc pas "vues" par le programme qui l'appelle.

On aura donc bien ainsi l'indépendance des différents modules de programme.

2. Passage des arguments

Il est évident qu'une procédure ne peut être un bloc isolé, sans lien avec les autres modules de programme; elle doit donc pouvoir recevoir des informations, et en retourner en échange; ceci constitue le problème du "passage des arguments".

Il faut donc préciser lors de l'appel de la procédure (instruction CALL) la liste des arguments, ou des "paramètres", à transmettre à la procédure (arguments "*effectifs*").

Lors de la définition de la procédure (instruction PROC), on écrira la liste des arguments "*formels*", recevant les valeurs des paramètres effectifs.

Il existe essentiellement deux méthodes classiques de passage d'arguments; nos instructions permettent effectivement l'utilisation des deux modes.

- **Passage par valeur**: ce sont alors simplement les valeurs des paramètres effectifs qui sont transmises à la procédure; on "recopie" pour cela les valeurs de ceux-ci après conversion éventuelle en cas de types différents, dans les variables formelles correspondantes.

Les valeurs en-tête d'un tableau sont, elles, directement recopiées dans le tableau formel, qui sera donc automatiquement créé avec la même structure que le tableau effectif; le nom devra être du même type que celui-ci, car il serait trop long à l'exécution de convertir un à un tous les éléments du tableau.

Ce mode est le passage d'arguments par défaut que nous avons retenu pour nos instructions, semblables en cela aux instructions correspondantes du PASCAL, de l'APL et des fonctions FN BASIC (ce mode n'existe pas en FORTRAN, mais il peut être simulé).

Dans ce mode de passage, aucune valeur n'est retournée vers le programme principal, qui est donc totalement "protégé" des traitements effectués dans la procédure.

- **Passage par adresse**: c'est alors l'adresse de l'argument réel qui est transmise au sous-programme lors de l'appel; lors du retour au programme principal, les valeurs des paramètres formels sont retournées vers celui-ci, c'est-à-dire que les paramètres effectifs sont modifiés.

On peut ainsi transmettre un nombre quelconque de résultats vers le programme principal, en prenant garde que certaines variables de celui-ci seront donc modifiées par l'exécution de la procédure.

Pour indiquer à l'interpréteur que l'on désire faire un passage par adresse, nous avons choisi d'utiliser ici, dans la liste des paramètres effectifs du CALL, le mot clé ARG ("argument").

Signalons que ce mode de passage est le mode de passage normal du FORTRAN; en PASCAL, le principe est le même qu'ici (utilisation du mot clé VAR); toutefois, cette déclaration doit être faite lors de la définition de la procédure et donc figée une fois pour toutes.

L'intérêt de faire plutôt la déclaration lors de l'appel est que la procédure pourra ainsi ne pas retourner les mêmes paramètres (et donc ne pas modifier toujours les mêmes paramètres effectifs) lors de deux appels différents: voir l'exemple du jeu des jetons donné plus loin.

II. L'instruction CALL

Elle réalise l'appel d'une procédure.

1. Syntaxe-action

On écrira :

CALL numéro de ligne (liste d'arguments)

$$\text{argument} = \left\{ \begin{array}{ll} \text{expression} & \text{ou DIM Nom de tableau} \\ \text{ARG variable} & \text{ou ARG DIM Nom de tableau} \end{array} \right.$$

La valeur de l'argument est transmise dans tous les cas à la procédure; si l'on écrit un nom de tableau précédé de DIM, c'est le tableau tout entier qui sera transmis.

Si l'on utilise la déclaration ARG, il y aura en plus retour de la valeur du paramètre formel correspondant; l'argument devra donc être alors une variable (ou un élément de tableau), ou un nom de tableau précédé de DIM.

L'instruction provoque le branchement à la ligne indiquée.

REMARQUES: Toutes les variables précédées de ARG devront avoir été définies auparavant, par l'affectation d'une valeur par exemple (mais pas en

l'écrivant dans une expression, qui ne crée pas les variables en mémoire : voir 2^e partie); on évite ainsi tout problème éventuel de déplacement de tableau en cas de création d'une nouvelle variable.

De même, les tableaux à transmettre devront avoir été définis auparavant par une instruction DIM normale, ceci car \$A48 appelée avec 1 dans \$6107 ne crée pas le tableau.

Bien entendu, tout manquement aux règles ci-dessus est détecté ou signalé, soit par "CN Error" (Can't continue) en cas de définition oubliée, soit par "SN Error" par exemple si on écrit ARG expression ou DIM nom (liste d'indices) comme argument.

2. La routine

La routine place d'abord dans la pile :

- l'adresse du premier caractère qui suit le CALL,
- le numéro de la ligne du CALL,
- la valeur &H3B, placée à l'adresse α

Ces 5 octets permettront le retour à l'instruction suivant le CALL, après exécution de la procédure.

Pour chaque argument, la routine empile ensuite successivement :

- l'adresse de la valeur de l'argument, ou celle de l'en-tête pour un tableau (précédé de DIM),
- le type de l'argument, les bits 7 et 6 étant mis à 1 pour un tableau ; dans le cas d'un tableau à retourner, précédé donc de ARG DIM, seul le bit 7 est mis à 1,
- la valeur de l'argument (sur un nombre d'octets égal au type), sauf en cas de tableau ou de déclaration ARG ; l'adresse empilée 3 octets auparavant est alors celle de cette valeur, c'est-à-dire qu'elle pointe dans la pile elle-même.

Après empilement du dernier argument, le sommet de la pile est à l'adresse β ; on empile alors :

- l'adresse contenue en \$611E,
- celle contenu en \$6120,
- l'adresse β ,

- l'adresse α du &H3B (les arguments formels sont donc décrits dans la pile entre $\alpha-1$ et β),
- la valeur &HC0 caractérisant le CALL.

Il y a alors branchement au numéro de ligne écrit après le CALL.

La routine utilise un sous-programme déterminant l'adresse et le type d'un argument; dans le cas d'un tableau, l'adresse retournée est celle du début de l'en-tête (voir 2^e partie).

Le listing est le suivant :

BB30	06	07	LDB	##07	14 octets→pile
BB32	BD	0336	JSR	\$0336	Débordement?
BB35	9E	22	LDX	\$22	Début zone libre
BB37	BF	BFFE	STX	\$BFFE	Adr. inutilisée
BB3A	109E	B9	LDY	\$B9	Adr. car. après CALL
BB3D	9E	2C	LDX	\$2C	N°ligne du CALL
BB3F	86	38	LDA	##3B	
BB41	34	32	PSHS	A,X,Y	Empilement
BB43	9D	B8	JSR	\$B8	Caractère courant
BB45	BD	06FD	JSR	\$06FD	Calcul adr.brancht.
BB48	BD	07E6	JSR	\$07E6	A-t'on "("?
BB4B	06	06	LDB	##06	11 octets maxi.
BB4D	BD	0336	JSR	\$0336	Débordement?
BB50	9D	B8	JSR	\$B8	
BB52	4C		INCA		A-t'on ARG (FFAB)?
BB53	27	21	BEQ	##B76	Par adresse
BB55	81	85	CMPL	##85	Code de DIM+1
BB57	26	06	BNE	##B5F	Variable
BB59	8D	58	BSR	##B83	S.P.+6 (tableau)
BB5B	8A	00	ORA	##C0	1→bits 7 et 6 type
BB5D	20	23	BRA	##B82	Suite
BB5F	BD	081A	JSR	\$081A	Calcul expression
BB62	96	05	LDA	\$05	Type
BB64	1F	89	TFR	A,B	Nbr.octets val.→B
BB66	08	03	ADDB	##03	
BB68	50		NEGB		
BB69	30	E5	LEAX	B,S	Adr.valeur→X
BB6B	34	12	PSHS	A,X	
BB6D	9F	3F	STX	\$3F	
BB6F	1F	14	TFR	X,S	Pour empiler valeur
BB71	BD	0737	JSR	\$0737	Valeur→pile
BB74	20	0E	BRA	##B84	Argument suivant
BB76	9D	B2	JSR	\$B2	2 ^d octet code
BB78	06	AB	LDB	##AB	Code de ARG

BB7A	90	D0	JSR	\$D0	A-t'on ARG ?
BB7C	80	2F	BSR	\$BBAD	Sous-programme
BB7E	27	02	BEQ	\$BB82	Suite (Variable)
BB80	8A	80	ORA	#\$80	Tableau (1→bit 7)
BB82	34	12	PSHS	A,X	
BB84	90	B8	JSR	\$B8	Caractère courant
BB86	81	29	CMPA	##29	Code de ""
BB88	27	04	BEQ	\$BB8E	Terminé
BB8A	90	CA	JSR	\$CA	A-t'on ", " ?
BB8C	20	BD	BRA	\$BB4B	Argument suivant
BB8E	DE	1E	LDU	\$1E	Début variables
BB90	109E	20	LDY	\$20	Début tableaux
BB93	1F	41	TFR	S,X	Adresse β→X
BB95	DC	8C	LDD	\$8C	Fond de la pile
BB97	83	0007	SUBD	##0007	Adr.α de #3B→D
BB9A	34	76	PSHS	A,B,X,Y,U	Empilement
BB9C	86	C0	LDA	##C0	
BB9E	34	02	PSHS	A	
BBA0	9E	22	LDX	\$22	Début zone libre
BBA2	BC	BFFE	CMPL	\$BFFE	S'est-il déplacé ?
BBA5	26	34	BNE	\$BBDB	Oui ⇒CN Error
BBA7	BD	0629	JSR	\$0629	Sur ligne indiquée
BBA9	7E	2AED	JMP	\$2AED	Exécution

Le sous-programme est le suivant:

BBAD	90	B8	JSR	\$B8	Caractère courant
BBAF	81	84	CMPL	##84	Code de DIM
BBB1	26	06	BNE	\$BBB9	Var.ou elt.de tab.
BBB3	86	01	LDA	##01	Tableau
BBB5	97	07	STA	\$07	Cherchera en-tête
BBB7	90	B2	JSR	\$B2	
BBB9	BD	0A48	JSR	\$0A48	Adr.→X;type→05
BBBC	06	07	LDB	\$07	
BBBE	27	06	BEQ	\$BBBC6	
BBC0	4D		TSTA		Tabl.existe déjà ?
BBC1	27	03	BEQ	\$BBBC6	oui
BBC3	7A	BFFE	DEC	\$BFFE	non
BBC6	96	05	LDA	\$05	Type
BBC8	F	07	CLR	\$07	
BBCA	50		TSTB		Var.ou tableau ?
BBCB	39		RTS		Retour

REMARQUES: On notera ici l'utilisation de deux octets situés en \$BFFE pour détecter tout déplacement de la zone des tableaux.

D'autre part, la routine \$A48 appelée avec 1 dans \$6107 retourne A=1 si le tableau correspondant n'existe pas en mémoire (et A=0 dans le cas contraire); on déplace alors \$BFFE, d'où la détection (à la fin) de l'erreur CN. La somme de contrôle est ici égale à 15720.

III. L'instruction PROC

Elle permet de définir les paramètres formels de la procédure; suivie du mot END, nous verrons qu'elle provoque le retour au programme principal.

1. Syntaxe-action

PROC (liste d'arguments formels)

Argument formel = Variable ou DIM Nom de tableau

L'argument formel prend dans tous les cas la valeur du paramètre effectif correspondant.

Dans le cas d'un nom de tableau précédé de DIM, le tableau formel est créé automatiquement avec la même structure que le tableau effectif; le type doit être alors le même que celui de ce tableau effectif.

Le nombre, la nature (variable simple ou tableau) et le type (numérique quelconque, ou chaîne) des paramètres formels et effectifs doit correspondre.

Tout manquement aux règles ci-dessus, ou par exemple la rencontre par l'interpréteur d'un PROC sans qu'il y ait eu appel par CALL, est signalé par un "CN (ou SN) Error".

REMARQUE: On peut écrire un élément de tableau comme argument; il y aura alors création automatique du tableau correspondant, de taille 11.

2. La routine

Elle vérifie que la valeur située en haut de la pile est bien &HC0 (précédé de l'adresse \$2B23 de retour à la boucle d'exécution des programmes: voir 2^e partie); elle remplace ensuite cette valeur par l'adresse du 1^{er} caractère

du 1^{er} argument (ceci pour le retour), empile la valeur &HC1 caractérisant PROC, et remplace l'adresse de retour au sommet.

Les zones des variables et des tableaux sont alors déplacées en zone libre (d'où l'indépendance totale entre les variables et tableaux de la procédure et celles du programme principal), puis les paramètres formels sont créés dans ces zones, et initialisés au fur et à mesure avec les valeurs des paramètres effectifs correspondants (pile balayée grâce au registre Y).

Il y a alors exécution de la procédure, jusqu'à l'instruction PROCEND.

Le listing est le suivant :

BBCC	81	80	CMFA	##80	Code de END
BBCE	27	6C	BEQ	\$BC3C	PROCEND⇒retour
BBD0	BD	07E6	JSR	\$07E6	A-t'on "("?
BBD3	35	40	FULS	U	Adresse \$2B23
BBD5	35	22	FULS	A,Y	Adr.a(CALL)→Y
BBD7	81	00	CMFA	##C0	
BBD9	27	05	BEQ	\$BBE0	
BBDB	06	11	LDB	##11	
BBDD	7E	0353	JMP	\$0353	CN Error
BBE0	4C		INCA		≠\$C1→A
BBE1	9E	B9	LDX	\$B9	Adr. 1 ^{er} caract.
BBE3	34	32	PSHS	A,X,Y	
BBE5	34	40	PSHS	U	Replace \$2B23
BBE7	9E	22	LDX	\$22	Adr.zone libre
BBE9	9F	1E	STX	\$1E	Déplact.zone var.
BBEB	9F	20	STX	\$20	Déplact.zone tab.
BBED	8D	BE	BSR	\$BBAD	Sous-programme
BBEF	27	21	BEQ	\$BC12	Variable
BBF1	EE	A3	LDU	,--Y	Tableau
BBF3	A8	A2	EORA	,-Y	Types égaux?
BBF5	84	BF	ANDR	##BF	o→bit 6
BBF7	4A		DECA		A-t'on \$80?
BBF8	28	E1	BVC	\$BBDB	CN Error
BBFA	BD	0ACB	JSR	\$0ACB	Nom→zone tableau
BBFD	EE	21	LDU	1,Y	Adr.tab.effectif
BBFF	1F	10	TFR	X,D	Adr.tab.formel→D
BC01	E3	04	ADD0	,U	D+taille tab.eff.→D
BC03	DD	22	STD	\$22	Déplact.zone libre
BC05	BD	033A	JSR	\$033A	Débordement mem.?
BC08	A6	00	LDR	,U+	1 octet tab.eff.
BC0A	A7	80	STA	,X+	→tab.formel
BC0C	9C	22	CMFX	\$22	fini?
BC0E	25	F8	BLO	\$BC08	Boucle
BC10	20	1E	BRA	\$BC30	Suite

<u>BC12</u>	9F	3F	STX	\$3F	Adr.var.form.
BC14	34	02	PSHS	A	Type var.form.→pile
BC16	AE	A3	LDX	,--Y	Adr.var.eff.→X
BC18	A6	A2	LDA	,--Y	Type var.eff.→A
BC1A	2B	BF	BMI	\$BBDB	CN Error(tableau)
BC1C	97	05	STA	\$05	
BC1E	9C	22	CMPX	\$22	Valeur dans pile?
BC20	25	02	BLO	\$BC24	non
BC22	1F	12	TFR	X,Y	Adr.valeur→Y
BC24	BD	0803	JSR	\$0803	Argu.eff.→AC flot.
BC27	35	02	PULS	A	Type var.form.
BC29	34	20	PSHS	Y	Sauvegarde de Y
BC2B	BD	0734	JSR	\$0734	Val.convertie→var.
BC2E	35	20	PULS	Y	
BC30	10AC	67	CMPY	7,S	Fin? (Y=β, de CALL)
BC33	27	04	BEQ	\$BC39	oui
BC35	9D	CA	JSR	\$CA	A-t'on ","?
BC37	20	B4	BRA	\$BBED	Argument suivant
BC39	7E	07E3	JMP	\$07E3	A-t'on ""? RTS

REMARQUE: Cette routine devra être implantée juste après CALL, ceci pour que les branchements relatifs au sous-programme et au CN Error soient exacts.

IV. L'instruction PROCEND

Elle provoque le retour au programme principal.

1. Syntaxe-action

PROCEND ou PROC END

Il y a retour à l'instruction du programme principal qui suit immédiatement le CALL.

Les paramètres formels correspondants aux paramètres effectifs précédés de ARG dans l'instruction CALL sont retournés au programme principal.

Une procédure peut contenir plusieurs PROCEND.

2. La routine

La routine dépile d'abord les boucles FOR ou WHILE non terminées, qui ne provoqueront donc pas d'erreur (routine \$2F3 : voir 2^e partie).

Puis elle vérifie la valeur située en haut de la pile (&HC1); les paramètres effectifs présents dans la pile sont examinés un à un (registre Y), et ils prennent éventuellement la valeur du paramètre formel correspondant (si leur adresse ne pointe pas dans la pile).

Les adresses des trois zones (variables, tableaux, libre) sont alors restaurées; toutes les variables et tableaux utilisés par la procédure ne seront donc pas "vues" par le programme principal.

Il y a enfin retour au programme principal, grâce à la routine \$64C (traitement de RETURN).

Le listing est le suivant :

BC3C	9D	B2	JSR	\$B2	1 ^{er} octet après END
BC3E	27	01	BEQ	\$BC41	
BC40	39		RTS		Provoque SN Error
BC41	86	FF	LDA	##FF	FOR,WHILE non...
BC43	97	3F	STA	\$3F	...dépile sinon
BC45	BD	02F3	JSR	\$02F3	Cherche FOR,WHILE
BC48	1F	14	TFR	X,S	Dépilement
BC4A	35	42	PULS	A,U	Adr.1 ^{er} arg.PROC→U
BC4C	81	C1	CMFA	##C1	
BC4E	27	03	BEQ	\$BC53	
BC50	7E	17A8	JMP	\$17A8	RE Error(Return er.)
BC53	DF	B9	STU	\$B9	Sur le 1 ^{er} arg. PROC
BC55	10AE	E4	LDY	,S	α(voir CALL)→Y
BC58	17	FF52	LBSR	\$BBAD	Sous-programme
BC5B	27	18	BEQ	\$BC75	Variable
BC5D	EE	A3	LDU	,--Y	Adr.tableau eff.
BC5F	A6	A2	LDA	,~Y	Type
BC61	84	40	ANDA	##40	Garde bit 6
BC63	26	29	BNE	\$BC8E	Suite (par valeur)
BC65	1F	10	TFR	X,D	par adr.;adr.déb.→D
BC67	E3	84	ADDD	,X	D+taille tab.→D
BC69	DD	57	STD	\$57	Adr.fin tableau
BC6B	A6	80	LDA	,X+	1 octet tab.form
BC6D	A7	C0	STA	,U+	→tableau eff.
BC6F	9C	57	CMFX	\$57	Fin?
BC71	25	F8	BLO	\$BC6B	Boucle
BC73	20	19	BRA	\$BC8E	Suite

BC75	EE	A3	LDU	,--Y	Adr.valeur var.
BC77	DF	3F	STU	\$3F	
BC79	1193	22	CMPIJ	\$22	Valeur dans pile?
BC7C	25	04	BLO	\$BC82	Non → transmettre val.
BC7E	1F	32	TFR	U,Y	On saute la valeur
BC80	20	0C	BRA	\$BC8E	Suite
BC82	BD	0803	JSR	\$0803	Val.form.→AC flot.
BC85	A6	A2	LDA	,--Y	Type arg.eff.
BC87	34	20	PSHS	Y	Sauvegarde de Y
BC89	BD	0734	JSR	\$0734	Val.convertie→var.
BC8C	35	20	PULS	Y	
BC8E	9D	B2	JSR	\$B2	
BC90	10AC	62	CMPIJ	2,S	Fin? (Y=β, de CALL)
BC93	26	C3	BNE	\$BC58	Argument suivant
BC95	35	76	PULS	A,B,X,Y,U	
BC97	1F	04	TFR	D,S	Pile restaurée
BC99	9E	1E	LDX	\$1E	Adresses zones...
BC9B	9F	22	STX	\$22	
BC9D	109F	20	STY	\$20	
BCA0	DF	1E	STU	\$1E	...restaurées
BCA2	A6	E4	LDA	,S	Valeur 3B→A
BCA4	7E	064C	JMP	\$064C	Trait. de RETURN

V. Applications simples

1. Mise en œuvre

Les instructions CALL, PROC et ARG seront implantées et mises en œuvre en ajoutant les instructions suivantes au programme de création :

```
2400 DATA CALL,&HBB30, liste des codes, FIN
2500 DATA PROC,&HBBCC, liste des codes, FIN
2990 DATA INC,&HBAB8,SWAP,&HBAFC,CALL,
      &HBB30,PROC,&HBBCC,ARG,&H7F3,FIN
```

Les adresses données ici sont indicatives ; rappelons toutefois que PROC devra être implanté immédiatement après CALL, et que l'adresse de ARG sera toujours \$7F3 (SN Error) puisque ARG n'est pas exécutable.

Il est enfin obligatoire d'écrire INC et SWAP en 2990 pour que ARG soit codé &HFFAB ; si on ne désire pas les créer, on n'écrira évidemment pas les lignes 2200 et 2300, et on écrira :

```
2990 DATA INC,&H7F3,SWAP,&H7F3,CALL,etc...
```

Après exécution, et sauvegarde (SAVEM) des octets correspondants et de la table des noms, les instructions seront activées par un simple EXEC &HBA01.

2. Premiers exemples

Exemple 1 :

```
90 REM Corps du Programme...
100 X=8.13:Y=-41.4:VAR=32.4
110 CALL 200("TEST",ARG X,Y+3.1,Y)
120 PRINT"Prém. Principal :";X;Y;VAR;Z;B%
130 END
200 'Procédure...
210 PROC(A$,VAR,Z,B%)
220 PRINT"Procédure   :";A$;VAR;Z;B%;X;Y
230 VAR=12.67:PROCEND
```

On obtient :

```
Procédure   :TEST 8.13 -38.3 -41  0  0
Prém. Principal : 12.67 -41.4  32.4  0  0
```

Les variables X et Y du programme ne sont pas "vues" par la procédure, et réciproquement pour les variables Z et B%.

La variable VAR de la procédure correspond à la variable X du programme principal, et sa valeur est retournée dans celle-ci ; elle n'a rien à voir avec la variable VAR du programme.

Exemple 2 :

```
100 N=20: DIM A$(N), B$(N)           'Taille N
110 'Lecture tableau A$ (à écrire...)
200 CALL 500(DIM A$, ARG DIM B$, N)
210 'On a dans B$ le tableau initial trié
220 'Suite (à écrire)
490 END
500 PROC(DIM T1$, DIM T2$, P)
510 'Tri dans T2$ du tableau T1$ (à écrire)
600 PROCEND
```

On peut bien sûr aussi trier directement dans le tableau A\$, en écrivant :

```
200 CALL 500(ARG DIM A$, N)
500 PROC(DIM T$, P)
```

3. La récursivité

Nous rappellerons ici qu'un sous-programme récursif est un sous-programme qui s'utilise lui-même dans sa propre définition.

Par exemple, une factorielle peut se définir par :

$$n! = n * (n-1)!$$

De même, la définition d'un nombre de Fibonacci est :

$$F_n = F_{n-1} + F_{n-2}$$

Enfin, un coefficient binominal peut se définir par :

$$C_n^P = C_{n-1}^{P-1} + C_{n-1}^P$$

Ces trois définitions sont récursives.

Le BASIC normal autorise une pseudo récursivité puisqu'un sous-programme peut s'appeler lui-même ; elle reste cependant très limitée à cause de la "globalité" des variables.

Nos procédures permettent par contre une totale récursivité, ce qui constitue un outil très puissant en programmation.

Exemple 1: n!

```
100 INPUT "Donner un nombre(1 a 33) ";N%
110 FACT=0:CALL 200(ARG FACT,N%)
120 PRINT "FACTORIELLE";N%;"=";FACT
130 END
195 'Procédure recursive...
200 PROC(F,N)      '...calcule F=N!
210 IF N<=1 THEN F=1 ELSE CALL 200(ARG F,N-1):F=F*N
220 PROC END
```

On obtient par exemple :

FACTORIELLE 4 = 24

Le fonctionnement est le suivant : il y a d'abord appel de PROC (F,4) par le programme principal ; la procédure elle-même appelle alors en 210 PROC(F,3), qui appelle elle-même PROC(F,2), qui appelle enfin PROC(F,1) où la règle d'arrêt (obligatoire !) est vérifiée.

Il y a alors retour de la valeur 1 vers PROC (F,2), qui retourne donc $1 \times 2 = 2$ vers PROC(F,3), qui retourne elle-même $2 \times 3 = 6$ vers PROC(F,4), retournant enfin $6 \times 4 = 24$ au programme principal.

A un moment donné de la récursion, il pourra y avoir par exemple une vingtaine d'appels imbriqués, chacun ayant créé ses propres variables, contenant toutes des valeurs différentes bien qu'ayant le même nom !

REMARQUE : FACT ="valeur" est obligatoire à la ligne 110 pour définir la variable FACT (précédée de ARG dans le CALL) ; par contre, en 210, F est déjà définie par l'instruction PROC elle-même.

Exemple 2: C_n^p

```
300 INPUT "N,P ";N,P
310 COMB=0:CALL 400(ARG COMB,N,P)
320 PRINT:PRINT P
330 PRINT N;"=";COMB;CHR$(13);
340 ATTRB 0,1:PRINT "C":ATTRB 0,0:END
400 'Procédure(recursive)
410 PROC(C,N,P):CP=0
420 IF P=0 OR N<=P THEN C=1 ELSE CALL 400(ARG C,N-1,P-1)
    :CALL 400(ARG CP,N-1,P):C=C+CP
430 PROCEND
```

On obtient par exemple :

$$C_{7}^3 = 35$$

On constatera qu'il est ici nettement plus facile d'écrire ou de comprendre le programme que de suivre exactement tous les appels et retours ! ceci est d'ailleurs fréquent avec la récursivité.

On notera aussi que les deux exemples ci-dessus pourraient être programmés sans utiliser la récursivité (ce qui serait d'ailleurs ici plus efficace) ; l'emploi de celle-ci permet par contre une écriture très claire et élégante.

Dans certains problèmes (parcours ou création "d'arbres", tri par "quicksort", etc...), elle est d'ailleurs quasi-indispensable, et doit être simulée au prix d'une gymnastique compliquée en cas d'utilisation de langages non récursifs (par exemple BASIC standard, ou FORTRAN).

VI. Application aux jeux

1. Le principe

Soit à programmer un jeu où le programme devra affronter un joueur.

La récursivité permet d'écrire très facilement un sous-programme analysant à un moment donné de la partie la situation de jeu, jusqu'à son terme.

A un instant donné de l'analyse, le sous-programme s'appelle lui-même pour tous les coups possibles de l'adversaire ; lorsqu'il y a ainsi arrivée à une situation finale perdante, la récursivité permet de revenir automatiquement à une situation antérieure ; dès qu'un coup gagnant est trouvé, il est retourné au programme principal.

2. Exemple: jeu des jetons

Dans ce jeu simple et classique, on part d'un nombre quelconque N de jetons ; le joueur à qui c'est le tour de jouer peut en prélever un ou plusieurs, sans en prendre plus du double de ce qui a été ôté par l'adversaire au coup précédent.

Le gagnant est celui qui ramasse le dernier jeton.

Exemple de partie :

11 jetons au départ :

- le programme en prend 3 ; restent 8 ; le joueur en prend 2,
- le programme en prend 1 ; restent 5 ; le joueur en prend 1 (forcé),
- le programme en prend 1 (forcé) ; restent 3 ; le joueur en prend 2,
- le programme prend le dernier et gagne.

A un moment donné de la partie, la situation de jeu est ici représentée par un couple (N,MAX), où N est le nombre de jetons et MAX le nombre maximal que l'on peut prendre.

Le listing du programme est donné ci-après ; on constatera le choix d'un coup aléatoire si N est supérieur à 19, ceci car la recherche de tous les coups possibles serait alors trop longue ; cela constitue donc la seule chance de battre le programme, imbattable autrement...

On remarquera enfin que la programmation de jeux tels que jeu des allumettes (jeu de Nim), tic-tac-toe, etc..., utiliserait exactement le même sous-programme ; seule changerait la représentation de la situation de jeu.

Le programme complet est le suivant :

```
10 CLEAR:&HBA00:PRINT"Lecture des routines de Procédure  
...."  
20 LOADM:EXEC&HBA01  
100 CLS:DEFINT A-Z:C=0:INPUT"Combien de Jetons au départ  
":N:MAX=N-1  
110 PRINT"Voulez vous commencer (O,N)?":A$=INPUT$(1):P  
RINT A$:PRINT  
120 IFA$="O" GOTO400  
190 '  
195 '...Boucle...  
200 COLOR4,3:PRINT"Je vais Prendre..."  
210 IFN<20 GOTO250  
220 FORI=1TO3000:NEXT:C=RND*2+.5:GOTO290  
250 CALL 1000(N,MAX,0,R,ARG C)  
290 BEEP:N=N-C:MAX=2*C:IF C>1 THEN A$="s" ELSE A$=""  
300 PRINT C:"Jeton":A$:"restent":N:COLOR4,6
```

```

310 IF N=0 THEN ATTRB 1,1:PRINT:PRINT"Vous avez Perdu..
... desole...":GOTO500
400 INPUT" Combien Prenez vous de Jetons";P
410 IF P<1 OR P>MAX THEN PRINT"FAUX":GOTO400
420 N=N-P:MAX=2*P:IF N>0 GOTO200 ELSE ATTRB1,1:PRINT:PR
INT"Vous avez gagne..... bravo..."
500 ATTRB0,0:PRINT:PRINT"Une autre Partie (O,N)?":A$=I
NPUT$(1)
510 IFA$="O"GOTO100 ELSE END
990 '
995 'ProcEDURE...
1000 PROC(N,M,JO,RES,CO)
1010 IF N<=M THEN RES=JO:CO=N:PROCEND
1020 FOR CO=M TO 1 STEP -1
1030 CALL1000(N-CO,CO+CO,1-JO,ARG RES,0)
1040 IF JO=RES THEN PROCEND ELSE NEXT
1050 RES=1-JO:CO=RND*2+.5:PROCEND

```

REMARQUE: On n'a ici besoin que des routines correspondant à CALL, PROC et PROCEND, qui pourront donc être seules implantées (à partir de \$BD40 par exemple, l'initialisation étant en \$BD01) et enregistrées sur cassette (de \$BD01 à \$BFFF) pour cette application.

Commentaires :

La variable JO contient 0 si c'est au programme de jouer (on a alors un coup gagnant si le CALL dans la boucle retourne RES=0, c'est-à-dire RES=JO), et 1 dans le cas contraire (on a alors un coup gagnant, pour le joueur donc, dès qu'on obtient un RES=1, c'est-à-dire RES=JO encore).

RES vaut 0 en cas de situation gagnante pour le programme, et 1 dans le cas contraire; en cas de situation gagnante pour l'un ou pour l'autre, on sort donc directement de la boucle avec RES=JO; on retourne par contre RES=1-JO en cas de sortie normale de celle-ci.

COUP est le coup gagnant (s'il existe) trouvé par le sous-programme.

On observera que toutes les parties possibles sont jouées et analysées par notre procédure, avec six instructions seulement !

On notera enfin que le CALL dans la procédure elle-même ne doit pas modifier le coup x envisagé, d'où la transmission par valeur de 0 par exemple; par contre, le coup x doit être retourné au programme principal, d'où une transmission par adresse dans celui-ci.

VII. Améliorations possibles

On pourrait par exemple créer assez facilement la possibilité de faire des appels de procédures *par nom*, et non plus par étiquette numérique.

Ceci améliorerait la clarté des programmes, et rendrait l'écriture de l'appel et de la définition de la procédure indépendante de la numérotation des lignes.

On pourrait du même coup créer des branchements *par labels* : l'instruction `BRANCH nom` provoquerait un branchement à la ligne commençant par `LABEL nom`, `LABEL` étant une instruction non exécutable ; son adresse de traitement serait donc (en `2990 DATA...`) `$663`, qui cherche le premier `Ø` ou `“ :` situé après le caractère courant.

Dans les deux cas (`CALL` et `BRANCH`), le nom serait traité par la routine `$A0A`, qui le rangerait en `$657A` (voir la routine `$A48`) ; le nom serait ensuite cherché dans le programme de la même manière qu'une étiquette numérique est cherchée par la routine `$4A0` (voir `GOTO`).

7

Les sprites

On désigne par "sprite" ("lutin" en français) un motif graphique programmable mobile sur l'écran.

Le déplacement s'effectue toujours sans modification ou effacement de la scène, c'est-à-dire que tout se passe comme si le sprite passait devant le décor; toutes les rencontres avec des objets fixes de la scène ou avec d'autres sprites sont détectées automatiquement.

L'utilisation de sprites facilite donc énormément l'écriture de programmes d'animation graphique; elle autorise de plus des animations de qualité: formes et couleurs entièrement programmables, mouvements continus à vitesse quelconque, etc...

La gestion des sprites est pratiquement toujours effectuée par un circuit spécialisé; celui-ci n'existant pas sur les TO7, nous présentons ici une méthode entièrement logicielle, se présentant sous la forme d'une routine de 340 octets; elle permet l'animation d'un nombre quelconque de sprites différents, choisis ici de taille 16 x 16 points (facilement modifiable).

I. Animation graphique classique

Lorsqu'on désire animer des objets graphiques sur l'écran, on doit effectuer les opérations suivantes :

1. Calcul à partir des coordonnées courantes (x_0 , y_0) de la nouvelle position (x_1 , y_1),
2. Effacement de l'objet, situé en (x_0 , y_0) (affichage d'un caractère "espace" ou d'un rectangle (BOXF) de couleur "fond"),
3. Dessin du nouvel objet, en (x_1 , y_1),
4. $x_1 \rightarrow x_0$; $y_1 \rightarrow y_0$
5. Retour au début.

On a ainsi un temps minimal entre l'effacement et le nouveau dessin ; on obtient donc un mouvement le moins "saccadé" et "clignotant" possible, mais non idéal dans tous les cas.

De plus, le déplacement de formes quelconques, non limitées à de simples rectangles, impose l'affichage (par LOCATE et PRINT) de caractères utilisateurs ; le mouvement ne peut donc s'effectuer que de 8 points en 8 points (25 lignes, 40 colonnes) et non de manière continue.

II. Instruction SPRITE

Elle permet d'éviter tous les inconvénients précédents, et elle accélère considérablement l'exécution de l'animation.

1. Syntaxe-action

Notre instruction peut se présenter sous deux formes différentes :

a) Forme normale :

SPRITE numéro – (x, y) [, couleur]

Le sprite dont le numéro est spécifié (de 0 à 15 ici) est effacé automatiquement (par restauration de la scène présente initialement à l'emplacement du sprite) s'il était actif, et immédiatement redessiné au nouvel emplacement (x, y); le décor présent à cet endroit est sauvegardé.

Le numéro peut être spécifié sous la forme d'une constante, d'une variable ou d'une fonction.

x et y désignent les coordonnées du coin supérieur gauche du sprite (0 à 319 pour l'abscisse x, 0 à 199 pour l'ordonnée y).

Si le paramètre "couleur" est absent, le dessin du sprite est effectué dans la couleur courante (modifiable par l'instruction COLOR); dans le cas contraire, c'est la couleur spécifiée qui est utilisée.

Enfin, dans le cas où le sprite recouvre au moins un point de "forme" (bit à 1 dans la mémoire écran "forme"), la mémoire d'adresse \$BFFD est mise à 0; elle contient -1(&HFF) dans le cas contraire.

b) Désactivation :

SPRITE numéro

Le sprite dont le numéro est spécifié est désactivé; il y a alors seulement effacement, toujours par restauration de la scène initiale.

La première utilisation d'un sprite réalise automatiquement l'activation de celui-ci; il n'y a alors pas d'effacement.

2. Dessin et définition d'un sprite

Chaque sprite comporte ici 16 × 16 points.

Le sprite de numéro NUM est défini par les 4 caractères utilisateurs de numéro $4 * \text{NUM} + 3$ à $4 * \text{NUM}$, un point "allumé" correspondant à un bit à 1.

GR\$(4 * NUM + 3) définit les quatre premières lignes du sprite, GR\$(4 * NUM + 2) et GR\$(4 * NUM + 1) les 8 suivantes, et enfin GR\$(4 * NUM) les 4 dernières.

Grâce à cette convention, les lignes de chaque sprite sont placées correctement en mémoire (après inversion en \$BD4B de chaque couple d'octets): voir la deuxième partie pour la représentation des caractères utilisateurs.

Exemple : Soit le petit hélicoptère suivant :

	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
FF																	F8
02																	00
1F																	C0
71																	E3
41																	FE
3F																	E3
0F																	C0
88																	42
7F																	FC

Le sprite correspondant, de numéro 2 par exemple, sera donc défini par les instructions suivantes :

```

10 CLEAR:&HB3FF,12
20 DEFGR$(11)=255,248,2,0,31,192,113,227
30 DEFGR$(10)=65,254,63,227,15,192,136,66
40 DEFGR$(9)=127,252,0,0,0,0,0,0
50 DEFGR$(8)=0,0,0,0,0,0,0,0

```

REMARQUE: On verra au paragraphe III que l'on peut associer 2 (ou plus) sprites pour créer des motifs de 32 x 16 points.

3. Gestion des sprites

Chacun des 16 sprites est géré de la manière suivante :

a) 48 octets contiennent les 16 lignes correspondant au motif du sprite, décalé à droite de *r* positions (*r* étant égal au reste de la division de l'abscisse *x* par 8, puisque la mémoire écran est constituée de segments de 8 points).

On obtient donc 3 octets pour chaque segment de 16 points du sprite, que l'on permutera avec les octets correspondants de la mémoire écran (mémoire de forme) pour dessiner le sprite, et pour permettre la restauration ultérieure de la scène initiale.

Ces 48 octets sont situés ici en \$B400 pour le premier sprite, en \$B460 pour le second, ..., jusqu'à \$B9A0 pour le 16^e.

REMARQUE: Si un des 3 octets d'un segment du sprite décalé ne contient aucun point de celui-ci, on le remplace par l'octet correspondant de la mémoire de forme; à cet emplacement l'écran ne sera donc pas modifié par le dessin du sprite, ce qui est normal.

b) 48 octets décrivent de même la couleur des 16 segments du sprite; ils sont eux aussi permutés avec les octets correspondants de la mémoire écran (mémoire de couleur) lors de l'affichage du sprite.

Ces 48 octets sont situés ici en \$B430 pour le premier sprite, en \$B490 pour le second, ..., jusqu'à \$B9D0 pour le 16^e.

REMARQUE: pour chaque octet, la couleur de forme (bits 3, 4, 5, et 6 pour le TO7-70) est celle du sprite, sauf s'il n'existe pas de points de celui-ci dans l'octet "forme" de l'écran à cet endroit; on prend alors bien sûr la couleur "forme" de l'écran à cet endroit.

La couleur de fond de chaque octet (bits 0, 1 et 2, et 7 pour le TO7-70) est celle de l'écran à l'endroit correspondant, sauf si l'octet U formé de la réunion de l'octet écran "forme" et du sprite décalé ne contient que des points de forme (U=&HFF); c'est alors la couleur *forme* de l'écran à cet endroit qui est prise comme couleur de fond, ce qui est là aussi normal (sprite passant dans une zone de couleur, dessinée par exemple par BOXF avec une couleur positive).

c) 4 octets contiennent respectivement l'abscisse et l'ordonnée du sprite; le premier octet de l'ordonnée (poids fort) non nul indique que le sprite est non actif.

Ces octets sont situés ici en \$BEC0 pour le premier sprite, \$BEC4 pour le second, ..., jusqu'à \$BEFC pour le 16^e.

4. La routine

Elle détermine d'abord les adresses des zones de gestion du sprite concerné, puis en \$BD19 celle du début des 4 caractères utilisateurs correspondant au sprite (utilisation de l'adresse contenue en \$612A: voir deuxième partie).

Après restauration de la scène présente initialement à l'emplacement du sprite (appel en \$BD34 du sous-programme de permutation entre la mémoire écran et la zone des 96 octets correspondants au sprite), l'image

du nouveau sprite est construite dans les 2 x 48 octets prévus pour l'affichage: en \$BD39, la forme est obtenue par décalage à droite des caractères utilisateurs correspondants; en \$BD84, la couleur est construite comme vu précédemment.

Il y a alors permutation en \$BDBA entre ces octets et la mémoire écran.

La routine est la suivante:

BCB3	86	FF	LDA	#\$FF	-1→A
BCB5	87	BFFD	STA	\$BFFD	Déteçtera collisions
BCB8	8D	0815	JSR	\$0815	Positionne sur n°
BCBB	8D	0770	JSR	\$0770	Traite n° sprite
BCBE	8D	0EBB	JSR	\$0EBB	Conversion en entier
BCC1	34	12	PSHS	A,X	N° et car. courant→S
BCC3	27	07	BEQ	\$BCCC	Désactivation
BCC5	108E	00C8	LDY	#\$00C8	200⇒hors écran
BCC9	8D	355C	JSR	\$355C	Traite coord.,couleur
BCCC	EC	61	LDD	1,S	N° sprite→D
BCCE	C1	0F	CMPB	#\$0F	≤15?
BCD0	23	03	BLS	\$BCD5	Oui
BCD2	7E	0B8C	JMP	\$0B8C	Non⇒FC Error
BCD5	58		LSLB		
BCD6	58		LSLB		Numéro *4→D
BCD7	C3	BEC0	ADDD	#\$BEC0	
BCDA	DD	63	STD	\$63	Adr. abscisse→\$63
BCDC	5C		INCB		
BCDD	5C		INCB		
BCDE	DD	65	STD	\$65	Adr. ordonnée→\$65
BCE0	A6	62	LDA	2,S	N° sprite→A
BCE2	C6	60	LDB	#\$60	
BCE4	3D		MUL		
BCE5	C3	B400	ADDD	#\$B400	
BCE8	DD	67	STD	\$67	Adr. 96 octets→\$67
BCEA	EC	9F6165	LDD	[\$6165]	Ordonnée précédente
BCEE	DD	5C	STD	\$5C→\$5C
BCF0	EC	9F6163	LDD	[\$6163]	Abscisse précédente
BCF4	DD	5A	STD	\$5A→\$5A
BCF6	35	12	PULS	A,X	Car.courant→A
BCF8	81	C8	CMPA	#\$C8	Doit-on redessiner?
BCFA	27	08	BEQ	\$BD04	Oui⇒suite
BCFC	17	00BF	LBSR	\$BD0E	Non⇒effacement
BCFF	6C	9F6165	INC	[\$6165]	Désactivation
BD03	39		RTS		Fin
BD04	FC	6576	LDD	\$6576	Nouvelle abscisse x
BD07	ED	9F6163	STD	[\$6163]	Reste de x/8
BD0B	C4	07	ANDB	#\$07	nbre.décalages+1

BD00	5C		INCB		
BD0E	D7	56	STB	\$56	
BD10	D7	57	STB	\$57	
BD12	FC	6578	LDD	\$6578	Nouvelle ordonnée y
BD15	ED	9F6165	STD	[\$6165]	
BD19	1F	10	TFR	X,D	N° sprite→D
BD1B	86	20	LDA	##20	
BD1D	3D		MUL		32*num.
BD1E	CB	21	ADDB	##21	
BD20	D3	2A	ADDD	\$2A	Début zone des GR\$
BD22	1F	03	TFR	D,U	(U-2) sur 1 ^{er} GR\$
BD24	BE	6576	LDX	\$6576	x→X
BD27	10BE	6578	LDY	\$6578	y→Y
BD2B	BD	F966	JSR	\$F966	TO7: adr.dans écran
BD2E	D	5C	TST	\$5C	Sprite actif?
BD30	26	07	BNE	\$BD39	Non⇒suite
BD32	34	50	PSHS	X,U	Sauvegarde
BD34	17	0087	LBSR	\$BD3E	Restaure décor...
BD37	35	50	PULS	X,U	...et efface sprite
BD39	9F	58	STX	\$58	Adr. dans mem. écran
BD3B	109E	67	LDY	\$67	Adr. zone forme→Y
BD3E	BD	F161	JSR	\$F161	TO7: mem.forme
BD41	86	10	LDA	##10	16 lignes à traiter
BD43	97	55	STA	\$55	
BD45	D6	56	LDB	\$56	
BD47	D7	57	STB	\$57	Nbre. décalages+1
BD49	EC	C3	LDD	,--U	1 segment GR\$→D
BD4B	1E	89	EXG	A,B	Dans bon ordre
BD4D	F	5A	CLR	\$5A	Pour décalage
BD4F	A	57	DEC	\$57	
BD51	27	06	BEQ	\$BD59	Décalage terminé
BD53	44		LSRA		Décalage (3 octets)
BD54	56		RORB		
BD55	6	5A	ROR	\$5A	
BD57	20	F6	BRA	\$BD4F	Boucle
BD59	ED	A4	STD	,Y	Range sprite décalé
BD5B	96	5A	LDA	\$5A	
BD5D	A7	22	STA	2,Y	
BD5F	C6	03	LDB	##03	3 octets à traiter
BD61	A6	A4	LDA	,Y	1 octet de forme
BD63	34	02	PSHS	A	
BD65	AA	84	ORA	,X	Sprite U écran→A
BD67	6D	A0	TST	,Y+	Octet sprite vide?
BD69	26	03	BNE	\$BD6E	Non⇒suite
BD6B	A7	3F	STA	-1,Y	Remplacé par écran
BD6D	4F		CLRA		
BD6E	A7	A82F	STA	47,Y	A→zone couleur
BD71	35	02	PULS	A	Forme sprite→A

BD73	A4	80	ANDA	,X+	Collision avec formes ?
BD75	27	03	BEQ	\$BD7A	Non
BD77	7F	BFFD	CLR	\$BFFD	Oui⇒0→\$BFFD
BD7A	5A		DECB		3 octets traités ?
BD7B	26	E4	BNE	\$BD61	Non⇒boucle
BD7D	30	8825	LEAX	37,X	ligne suivante écran
BD80	A	55	DEC	\$55	Terminé ?
BD82	26	C1	BNE	\$BD45	Non⇒ligne suivante
BD84	86	10	LDA	##10	16 lignes à traiter
BD86	97	55	STA	\$55	...pour couleur
BD88	9E	58	LDX	\$58	
BD8A	7A	E7C3	DEC	\$E7C3	Sélect.mem.couleur
BD8D	B6	6038	LDA	\$6038	Couleur sprite...
BD90	48		LSLA		
BD91	48		LSLA		
BD92	48		LSLA		
BD93	97	56	STA	\$56	→bits 5,4,3(forme)
BD95	06	03	LDB	##03	
BD97	A6	80	LDA	,X+	3 octets à traiter
BD99	6D	A4	TST	,Y	Couleur écran→A
BD9B	27	11	BEQ	\$BD4E	
BD9D	84	C7	ANDA	##C7	Pas de pts.du sprite
BD9F	9A	56	ORA	\$56	Garde couleur fond
BDA1	6C	A4	INC	,Y	Couleurs fond+forme
BDA3	26	09	BNE	\$BD4E	A-t'on U=FF ?
BDA5	A6	1F	LDA	-1,X	Non⇒terminé
BDA7	84	38	ANDA	##38	Couleurs écran→A
BDA9	44		LSRA		Garde couleur forme
BDAA	44		LSRA		
BDA8	44		LSRA		
BDAC	9A	56	ORA	\$56	→bits 0,1,2(fond)
BD4E	A7	A0	STA	,Y+	Couleurs fond+forme
BDB0	5A		DECB		Rangement couleurs
BDB1	26	E4	BNE	\$BD97	3 octets traités ?
BDB3	30	8825	LEAX	37,X	Non⇒boucle
BDB6	A	55	DEC	\$55	ligne suivante écran
BDB8	26	0B	BNE	\$BD95	16 lignes traitées ?
BDBA	9E	58	LDX	\$58	Non⇒ligne suivante
BDBC	20	0A	BRA	\$BDC8	Oui
BDBE	9E	5A	LDX	\$5A	Permut.écran-zone
BDC0	109E	5C	LDY	\$5C	Abscisse précédente
BDC3	BD	F966	JSR	\$F966	Ordonnée précédente
BDC6	9F	58	STX	\$58	T07 :adr. dans écran
BDC8	109E	67	LDY	\$67	
BDCB	BD	F161	JSR	\$F161	Adr.zone 96 octets
BDCE	86	10	LDA	##10	T07 :mem.forme
BDD0	97	55	STA	\$55	16 lignes à permuter
BDD2	EE	84	LDU	,X	

2 octets écran

BDD4	EC	A4	LDD	,Y	2 octets sprite
BDD6	EF	A1	STU	,Y++	
BDD8	ED	84	STD	,X	
BDDA	A6	02	LDA	2,X	3° octet écran
BDDC	E6	A4	LDB	,Y	3° octet sprite
BDD E	A7	A0	STA	,Y+	
BDE0	E7	02	STB	2,X	
BDE2	30	8828	LEAX	40,X	Ligne suivante écran
BDE5	8C	5F40	CMPL	##5F40	Sort-on de l'écran?
BDE8	24	04	BHS	\$BDEE	Oui⇒terminé
BDEA	A	55	DEC	\$55	16 lignes traitées?
BDEC	26	E4	BNE	\$BDD2	Non⇒boucle
BDEE	B6	E7C3	LDA	\$E7C3	
BDF1	44		LSRA		Bit0→C
BDF2	24	07	BHS	\$BDFB	C=0⇒terminé
BDF4	7A	E7C3	DEC	\$E7C3	Sélect.mem.couleur
BDF7	9E	58	LDX	\$58	
BDF9	20	D3	BRA	\$BDCE	Perm.coul.écran-zone
BDFB	39		RTS		

REMARQUE: La routine devra être légèrement modifiée pour le TO7-70: adresses de gestion déplacées éventuellement, adresses \$F966 et \$F161 du moniteur à remplacer par \$EF15 et \$F328: \$F966 place dans X l'adresse de l'octet de la mémoire écran contenant le point de coordonnées (X, Y); \$F161 envoie 1 dans le bit 0 de \$E7C3, ce qui sélectionne la mémoire de forme: voir 2° partie).

5. Mise en œuvre

Tous les sprites doivent être désactivés au départ; ceci sera réalisé en ajoutant une instruction JSR DESACT dans le sous-programme d'initialisation, DESACT(\$BCA8 ici) étant l'adresse d'une petite routine initialisant les poids forts des ordonnées (\$BEC2, BEC6, ..., BEFE) avec une valeur non nulle.

On utilisera pour cela les instructions suivantes:

BCA8	8E	BEC2	LDX	##BEC2	Ordonnée 1 ^{er} sprite
BCAB	86	1F	LDA	##1F	Pour compter
BCAD	A7	81	STA	,X++	
BCAF	4A		DECA		Terminé?
BCB0	26	FB	BNE	\$BCAD	Non⇒continuer
BCB2	39		RTS		Oui

On ajoutera donc au programme de création les lignes suivantes :

```
1715 DATA BD,BCA8 'Desactivation sSprites
2600 DATA SPRITE,&HBCA8,8E,BEC2,86,1F,A7,81,4A,26,FB,39
2610 DATA 86,FF,B7,.....,20,D3,39,FIN SPRITE
```

Enfin, on ajoutera en 2990 le nom (SPRITE) et l'adresse (\$BCB3) de la routine.

6. Exemple

Après sauvegarde (SAVEM "ROUT", &HBA01, &HB FCC, 0) de la routine accompagnée de l'initialisation et des tables de noms, on pourra animer l'hélicoptère précédent par le petit programme suivant :

```
400 CLEAR,&HB3FF,12:LOADM "ROUT":EXEC&HBA01
410 DEFGR$(11)=255,248,2,0,31,192,113,227
420 DEFGR$(10)=65,254,63,227,15,192,136,66
430 DEFGR$(9)=127,252,0,0,0,0,0,0
450 DEFGR$(8)=0,0,0,0,0,0,0,0
500 Y=RND(1)*185:C=RND(1)*8
510 FOR X=304 TO 0 STEP -2
520 SPRITE 2-(X,Y),C
530 IF PEEK(&HBFFD)=0 THEN BEEP 'Collision
540 FOR I=1 TO 30:NEXT 'Ralentissement
550 NEXT:GOTO500
```

On constatera que l'hélicoptère se déplace sans effacer les formes qu'il recouvre dans son mouvement; toutes les "collisions" avec l'une d'entre elles provoquent ici l'émission d'un "bip" sonore, bien sûr sans que le mouvement soit stoppé.

Enfin, la couleur du sprite change à chaque trajet.

III. Application aux jeux d'action

Nous donnons ici un exemple simple, que l'on pourra compléter à loisir: sonorisation, dessin de nuages dans le ciel, effet d'explosion, contrôle du

second hélicoptère par un second joueur (voir la routine transformant le clavier, dans la 4^e partie), etc...

Le jeu consiste à lâcher une bombe (par appui sur une touche quelconque du clavier), qui tombe en chute libre selon la parabole normale; elle doit atteindre l'hélicoptère qui passe en sens inverse dans le bas de l'écran.

Le programme utilise 4 sprites différents, le 1^{er} et le 4^e étant associés pour créer un motif (hélicoptère) de 32 x 16 points; on notera à la ligne 270 la désactivation initiale du 1^{er} sprite, ce qui permet une animation correcte du motif.

Le programme est le suivant:

```
10 CLEAR,&HB3FF,16:LOADM:EXEC&HBA01:CLS
20 DEFGR$(3)=7,255,7,255,0,0,0,0
30 DEFGR$(2)=0,15,192,15,240,63,112,63
40 DEFGR$(1)=63,255,63,255,112,63,240,63
50 DEFGR$(0)=192,15,0,15,0,12,0,12
60 DEFGR$(7)=14,0,3,0,3,254,15,255
70 DEFGR$(6)=3,254,3,0,14,0,0,0
72 DEFGR$(5)=0,0,0,0,0,0,0,0
74 DEFGR$(4)=0,0,0,0,0,0,0,0
80 DEFGR$(11)=255,248,2,0,31,192,113,227
90 DEFGR$(10)=65,254,63,227,15,192,136,66
100 DEFGR$(9)=127,252,0,0,0,0,0,0
110 DEFGR$(8)=0,0,0,0,0,0,0,0
120 DEFGR$(15)=255,255,255,255,112,0,112,0
130 DEFGR$(14)=255,192,255,192,192,248,192,252
140 DEFGR$(13)=192,12,192,12,255,248,255,240
150 DEFGR$(12)=255,192,255,192,0,192,0,192
190 SC=0:FOR NTIR= 0 TO 19
200 A$=INKEY$:SPRITE0=(0,50):CLS
210 BEEP:LOCATE0,0,0:COLOR0,3:PRINT"SCORE  :",SC:"SUR:",N
TIR:COLOR4,6
220 YJ=80*RND(1)+10
230 YC=RND(1)*50+130
240 YB=YJ+16:TIR=0
250 FOR XJ=0 TO 288 STEP 3+RND(1)/2
260 XJ2=XJ+16:IF TIR=0 THEN TIR=-((INKEY$<>"")*.08 ELSE
YB=YB+TIR:TIR=TIR+.16
270 SPRITE0:SPRITE3=(XJ2,YJ):SPRITE0=(XJ,YJ):SPRITE1=(X
J+9,YB)
280 IF PEEK(&HBFFD) THEN SPRITE2=(304-XJ,YC),1:NEXT XJ
290 SPRITE1:SPRITE2:SPRITE3:CLS
300 IF YB>179 OR YB<110 GOTO 340
305 IF XJ>288 GOTO 340
```

```

310 SCREEN0,1,1:SC=SC+1
315 PRINT:PRINT:ATTRB1,1:PRINT"TOUCHE...":ATTRB0,0
320 BEEP:FOR I=0 TO 1500:NEXT
330 SCREEN4,6,6
340 NEXT NTIR
350 BEEP:LOCATE0,0,0:PRINT"SCORE :";SC;"SUR";NTIR
360 PRINT:PRINT:INPUT"UNE AUTRE PARTIE(O/N) ";A$
370 IF A$="O" GOTO190
380 END

```

8

Résumé de la troisième partie

Nous avons donné un programme BASIC permettant de créer un nombre quelconque de fonctions ou instructions nouvelles, que l'on peut utiliser ensuite exactement comme celles d'origine.

Nous avons présenté alors un certain nombre d'applications permettant d'utiliser :

- les fonctions FNR et FND de conversion degrés-radians,
- les boucles WHILE...WEND,
- l'instruction INC d'incrémentation rapide,
- l'instruction SWAP permutant les contenus de deux variables,
- les instructions CALL, PROC et ARG permettant d'écrire des procédures à variables (ou tableaux) locales, la récursivité étant permise,
- l'instruction SPRITE permettant d'animer des objets programmables entièrement définissables (taille, couleur, motif).

Nous donnons ci-après le listing de tous les DATA correspondants, à écrire dans le programme de création.

Rappelons que l'on pourra bien sûr écrire d'autres instructions, en s'inspirant des mêmes principes; on veillera particulièrement à prévoir toutes les possibilités de mauvaise écriture, ou de mauvais emploi, afin d'éviter tout blocage possible du programme BASIC utilisant ces nouvelles instructions.

Pour ceux qui voudront adapter à d'autres ordinateurs les notions présentées ici, nous nous sommes efforcés d'être le plus clair et le plus général possible; c'est ainsi que pour chaque application, nous avons donné le principe général, puis des explications sur la réalisation et le fonctionnement de la routine; celle-ci a ensuite été commentée instruction par instruction.

Nous pensons donc que l'adaptation se révélera rapidement simple.

Listing des DATA:

```
1700 DATA Initialisations,&HBA01,CC,BA40,FD,623D
1710 DATA CC,BA70,FD,6234,C3,01E,FD,6237
1715 DATA BD,BCA8
1720 DATA CC,BF19,FD,6207,B6,BF00,B7,6206,S6,7E,B7,6273
,1F,50,C3,007,FD,6274,39
1730 DATA 8E,BF01,80,A7,2A,3,7E,7F3,7E,2B38,FIN
1990 '
1995 'Routines
2000 DATA Fonctions,&HBA40,9D,B2,34,2,9D,B2,BD,7DE,BD,2
52B,BD,1C81,CC,7B8E,DD,55,F,5D,CC,FA35,DD,57,35,2,81,52
,26,3,7E,25B6,81,44,26,3,7E,920,7E,7F3,FIN
2095 '
2100 DATA WHILE-WEND,&HBA70,32,62,C6,4,BD,3,36,DE,B9,9E
,2C,34,50,BD,16AC,9E,78,9F,2C,9D,B2,DE,B9,86,AF,34,42,8
D,0
2110 DATA 32,62,AE,61,9C,B9,27,5,C6,19,7E,353,AE,65,9F,
B9,BD,81A,BD,1CC8,27,7,AE,63,9F,2C,7E,2AED,AE,61,32,67,
7E,1669,FIN WEND
2195 '
2200 DATA INC,&HBAB8,BD,A48,9F,3F,9D,B8,26,11,8E,2380,9
D,CD,2A,3,8E,F9CA,BD,1C08,C,55,20,E,96,5,34,2,9D,CA,BD,
81A,35,2,BD,2510,9E,3F,BD,24FD,29,A,24,6,AE,84,9F,65,20
,5,C,3,BD,1ABF,BD,2590,7E,1C36,FIN
2295 '
2300 DATA SWAP,&HBADC,BD,800,32,78,10DF,3F,96,5,9E,3D,3
4,12,BD,737,9D,CA,BD,800,96,5,97,42,35,12,9F,3F,BD,734,
1F,41,BD,803,96,42,9E,3D,9F,3F,BD,734,32,68,39,FIN
2395 '
```

```

2400 DATA CALL,&HBB30,C6,7,BD,336,9E,22,BF,BFFE,109E,B9
,9E,2C,86,3B,34,32,9D,B8,BD,6FD,BD,7E6,C6,6,BD,336,9D,B
8,4C,27,21,81,85,26,6,8D,58,8A,C0,20,23,BD,81A,96,5,1F,
89,CB,3,50,30,E5,34,12,9F,3F,1F,14,BD,737,20,E
2405 DATA 9D,B2,C6,AB,9D,D0,8D,2F,27,2,8A,80,34,12,9D,B
8,81,29,27,4,9D,CA,20,BD,DE,1E,109E,20,1F,41,DC,8C,83,0
07,34,76,86,C0,34,2,9E,22,BC,BFFE,26,34,BD,629,7E,2AED
2410 DATA 9D,B8,81,84,26,6,86,1,97,7,9D,B2,BD,A48,D6,7,
27,6,4D,27,3,7A,BFFE,96,5,F,7,5D,39,FIN CALL
2500 DATA PROC,&HBBCC,81,80,27,6C,BD,7E6,35,40,35,22,81
,C0,27,5,C6,11,7E,353,4C,9E,B9,34,32,34,40,9E,22,9F,1E,
9F,20,8D,BE,27,21,EE,A3,A8,A2,84,BF,4A,28,E1,BD,ACB,EE,
21,1F,10,E3,C4,D0,22,BD,33A
2510 DATA A6,C0,A7,80,9C,22,25,F8,20,1E,9F,3F,34,2,AE,A
3,A6,A2,2B,BF,97,5,9C,22,25,2,1F,12,BD,803,35,2,34,20,B
D,734,35,20,10AC,67,27,4,9D,CA,20,B4,7E,7E3
2520 DATA 9D,B2,27,1,39,86,FF,97,3F,BD,2F3,1F,14,35,42,
81,C1,27,3,7E,17A8,DF,B9,10AE,E4,17,FF52,27,18,EE,A3,A6
,A2,84,40,26,29,1F,10,E3,84,D0,57,A6,80,A7,C0,9C,57,25,
F8,20,19,EE,A3,DF,3F,1193,22,25,4,1F,32,20,C,BD,803,A6,
A2,34,20,BD,734,35,20
2530 DATA 9D,B2,10AC,62,26,C3,35,76,1F,4,9E,1E,9F,22,10
9F,20,DF,1E,A6,E4,7E,64C,FIN PROC
2595 '
2600 DATA SPRITE,&HBCA8,8E,BEC2,86,1F,A7,81,4A,26,FB,39
2610 DATA 86,FF,B7,BFFD,BD,815,BD,770,BD,EBB,34,12,27,7
,108E,0C8,BD,355C,EC,61,C1,F,23,3,7E,88C,58,58,C3,BEC0,
DD,63,5C,5C,DD,65,A6,62,C6,60,3D,C3,B400,DD,67
2620 DATA EC,9F6165,DD,5C,EC,9F6163,DD,5A,35,12,81,C8,2
7,8,17,0BF,6C,9F6165,39
2630 DATA FC,6576,ED,9F6163,C4,7,5C,D7,56,D7,57,FC,6578
,ED,9F6165,1F,10,86,20,3D,CB,21,D3,2A,1F,3,BE,6576,10BE
,6578,BD,F966,D,5C,26,7,34,50,17,087,35,50,9F,58,109E,6
7,BD,F161,86,10,97,55
2640 DATA D6,56,D7,57,EC,C3,1E,89,F,5A,A,57,27,6,44,56,
6,5A,20,F6,ED,A4,96,5A,A7,22,C6,3,A6,A4,34,2,AA,84,6D,A
0,26,3,A7,3F,4F,A7,A82F,35,2,A4,80,27,3,7F,BFFD,5A,26,E
4,30,8825,A,55,26,C1
2650 DATA 86,10,97,55,9E,58,7A,E7C3,B6,6038,48,48,48,97
,56,C6,3,A6,80,6D,A4,27,11,84,C7,9A,56,6C,A4,26,9,A6,1F
,84,38,44,44,44,9A,56,A7,A0,5A,26,E4,30,8825,A,55,26,DB
,9E,58,20,A
2660 DATA 9E,5A,109E,5C,BD,F966,9F,58,109E,67,BD,F161,8
6,10,97,55,EE,84,EC,A4,EF,A1,ED,84,A6,2,E6,A4,A7,A0,E7,
2,30,8828,8C,5F40,24,4,A,55,26,E4,B6,E7C3,44,24,7,7A,E7
C3,9E,58,20,D3,39,FIN SPRITE
2950 DATA FIN des routines
2990 DATA INC,&HBAB8,SWAP,&HBAFC,CALL,&HBB30,PROC,&HBBC
C,ARG,&H7F3,SPRITE,&HBCB3,FIN

```

Résultat de l'exécution du programme de création :

ROUTINE Initialisations		
Debut:BA01	Fin:BA3C	Somme: 7153
ROUTINE Fonctions		
Debut:BA40	Fin:BA6D	Somme: 4888
ROUTINE WHILE-WEND		
Debut:BA70	Fin:BA84	Somme: 7613
ROUTINE INC		
Debut:BA88	Fin:BAF9	Somme: 6044
ROUTINE SWAP		
Debut:BAFC	Fin:BA2D	Somme: 4245
ROUTINE CALL		
Debut:BB30	Fin:BB0B	Somme: 15720
ROUTINE PROC		
Debut:BBCC	Fin:BCA6	Somme: 22239
ROUTINE SPRITE		
Debut:BCA8	Fin:BDFB	Somme: 38613
Implantation terminée...		
Enregistrer de BA01 à BFCC (Par SAVEM)		

Cas du T07-70 :

Les valeurs soulignées seront remplacées par EA24 à la ligne 2200, et par EF15 et F328 aux lignes 2630 et 2660: voir pages 110 et 141.

REMARQUE: Toutes les adresses d'implantation données en tête des routines seront changées dans le cas du T07 de base (fin de la RAM en \$7FFF), et éventuellement aussi pour le T07-70 si on veut bénéficier de toute la mémoire disponible (fin en \$DFFF).

On veillera alors bien sûr à modifier la ligne 2990, ainsi que les adresses écrites dans le sous-programme d'initialisation (et en particulier aux lignes 1720 et 1730: voir les remarques du chapitre IV) et dans les routines correspondant à CALL et SPRITE.

AMÉLIORER LES PERFORMANCES

Nous avons vu dans la 2^e partie qu'à chaque rencontre par l'interpréteur d'un nom de variable ou de tableau, il y a "balayage" des zones correspondantes jusqu'à trouver le nom en question.

Il en est de même pour les étiquettes de branchement, recalculées et recherchées dans le programme à chaque fois.

Il en résulte évidemment une chute des performances par rapport à celles qu'offrirait un compilateur, où tout ce travail (et toute la traduction en langage machine) est effectué une fois pour toutes, préalablement à l'exécution.

Par contre, ce mode de traitement permet un travail "conversationnel" extrêmement souple et agréable, puisque les modifications d'un programme en cours de mise au point sont prises en compte immédiatement, chose évidemment impossible avec un compilateur.

Toutefois, un programme ayant été mis au point, il devient complètement inutile de bénéficier de la possibilité précédente.

Nous proposons donc une méthode permettant de compiler au fur et à mesure de l'exécution les adresses des variables, ainsi que les branchements et les valeurs des constantes.

Notre méthode opère automatiquement à partir du programme écrit normalement, et c'est en principe sous cette forme que celui-ci sera conservé sur bande ou disquette; la compilation n'est mise en œuvre qu'après exécution d'un sous-programme d'initialisation, le programme pouvant bien sûr être modifié à volonté auparavant.

On constate donc que les contraintes apportées par cette compilation "interactive" sont minimales; le résultat est une vitesse d'exécution pratiquement doublée.

Toute application qui "tourne" verra donc ses performances très nettement améliorées par l'application de notre méthode, mise en œuvre par un simple EXEC écrit au début du programme.

Nous parlerons aussi dans cette partie de l'adaptation (toujours possible) de cette méthode à d'autres micro-ordinateurs que les TO7.

Enfin, nous commencerons à voir comment modifier l'action du clavier des TO7, dont le mode normal de fonctionnement est tout à fait inadapté à certains cas (jeux en particulier).

Modification du fonctionnement du clavier

Tous les jeux opérant en "temps réel" se présentent sous la forme d'une boucle, dans laquelle on lit le clavier ou les manettes de jeu ; en fonction du résultat de la lecture, une action (déplacement d'un objet par exemple) est effectuée et ses conséquences analysées (collision ou non...), après quoi il y a retour au début de la boucle.

Le clavier des TO7 est malheureusement très mal adapté à ce type de fonctionnement puisqu'en cas d'appui continu sur une touche, il y a en permanence décodage de celle-ci, ce qui provoque à chaque fois une temporisation et l'émission d'un "bip" sonore ; cela est désagréable, et ralentit surtout énormément le déroulement du programme.

Nous allons voir qu'il est facile de supprimer cet inconvénient, à partir des observations de la deuxième partie.

1. Suppression de l'action du clavier

Il est possible de supprimer les "bips" émis en rafale en cas d'appui continu sur une touche en intervenant sur le registre de contrôle \$E7C1 du

port P du circuit 6846 du TO7 ; il suffit en effet de faire pour cela *POKE &HE7C1,0* ; réciproquement, *POKE &HE7C1,48* permet de revenir à la situation normale.

Pour le TO7-70, il suffit de mettre à 1 (par *POKE &H6073,1*) le registre BUZZ du moniteur.

Ceci ne résout pas notre problème, puisqu'il y a toujours décodage des touches appuyées et temporisation, d'où ralentissement très sensible de l'exécution.

Or, on a vu dans la 2^e partie qu'à chaque nouvelle instruction du BASIC, il y a appel en \$2AF0 de la routine \$32BB de surveillance du clavier ; celle-ci commence par un appel de \$6294, où on trouve 3 octets disponibles (voir 3^e partie), le premier contenant RTS.

En \$6294, la pile S contient donc l'adresse \$2AF3 (retour de \$32BB), puis \$32BE au sommet (retour de \$6294) ; si on dépile cette dernière adresse, un RTS provoquera le retour directement en \$2AF3, sans exécution de la routine \$32 BB.

Pour supprimer la surveillance du clavier à chaque instruction BASIC exécutée, il suffit donc d'écrire en début de programme :

```
POKE &H6295,98:POKE &H6296,57:POKE &H6294,50
```

Ceci correspond en effet à LEAS 2,S/RTS.

On remarquera que ceci ne modifie en rien le fonctionnement des instructions INPUT et de la fonction INKEY\$, qui restent donc parfaitement utilisables.

REMARQUE 1 : Il faut modifier \$6294 en dernier et non en premier, car il y a appel de \$6294 entre chaque POKE, d'où "plantage" si on commence à implanter le LEAS avant d'avoir implanté la partie adresse.

REMARQUE 2 : Les touches STOP et CNT/C ne provoqueront bien sûr plus l'arrêt du programme, qui devra être éventuellement stoppé par la touche "Initialisation" ; ceci ne comporte aucun inconvénient (taper le "1" du menu pour revenir au programme).

Pour revenir au fonctionnement normal du clavier, on fera un simple :

```
POKE &H6294,&H39 (code de RTS)
```

Exemple d'application

Dans la boucle d'un programme temps réel, un INKEY\$ ou INPUT\$(n) ou INPUT fonctionnera exactement comme d'habitude, mais le clavier ne sera lu que lors de l'instruction correspondante.

En cas d'appui continu sur une touche (par exemple pour un jeu devant déplacer un mobile : raquette, ou vaisseau spatial, etc...), il n'y aura donc plus le ralentissement de l'exécution, et l'émission des "bips" en rafale.

Nos seuls trois POKE rendront donc beaucoup plus agréable l'utilisation de tous les programmes de ce type.

II. Jeux à un ou deux joueurs par le clavier

1. Le problème

Nous allons traiter le cas où l'on désire pouvoir déplacer un mobile à partir du clavier, 2 joueurs pouvant jouer en même-temps, l'un avec les touches de déplacement du curseur, l'autre avec les touches A,Q,S et W situées sur la gauche du clavier.

Pour jouer à partir des touches de déplacement du curseur, on doit normalement écrire :

```
A$=INKEY$:IF A$="" GOTO α1  
ON ASC(A$)-7 GOTO α2,α3,α4,α5
```

Ceci ne permet pas les déplacements en diagonale et le jeu à deux joueurs, et est sensiblement plus lent que le simple :

```
ON STICK(I) GOTO...
```

que l'on peut utiliser avec les manettes de jeu.

Nous allons donc écrire en langage machine une routine appelée par USRØ et retournant exactement les mêmes valeurs que l'instruction STICK ; on pourra donc remplacer toute utilisation de STICK par le simple mot USRØ, permettant de se passer des manettes de jeu, avec un fonctionnement et une rapidité exactement équivalents (au prix il est vrai d'une manipulation un peu moins agréable).

Les déplacements en diagonale seront simplement obtenus par l'appui simultané sur 2 touches ; par exemple, ↑ et →, ou A et S, correspondra à un déplacement selon ↗, codé 2.

L'appel du USR0(0) correspondra au décodage des 4 touches de déplacement du curseur ; l'appel de USR0(x) avec x≠0 décodera les touches A,Q,S et W logiquement disposées, c'est-à-dire A correspondant à ↑, Q à ←, S à → et W à ↓.

Les valeurs retournées (les mêmes que par STICK) sont les suivantes :



2. La routine

Le décodage des touches utilise l'organisation matricielle du clavier, vue dans la 1^{re} partie ; si on place par exemple la valeur &HFB en \$E7C9, le bit 3 de \$E7C8 sera à 0 si la touche Q est appuyée, etc...

La routine est la suivante :

BD30	BD	24C3	JSR	#24C3	Conversion argument
BD33	86	FE	LDA	##FE	0←bit0(T07)
BD35	B7	E7C9	STA	#E7C9	
BD38	CC	020B	LDD	##020B	2←FD
BD3B	8E	0000	LDX	##0000	
BD3E	D	58	TST	#58	Joueur à droite ?
BD40	26	08	BNE	#BD4A	A,Q,W ou S
BD42	8D	32	BSR	#BD76	Si ↑,9→X
BD44	8D	30	BSR	#BD76	Si ←,7→X
BD46	8D	2E	BSR	#BD76	Si ↓,5→X
BD48	20	10	BRA	#BD5A	lecture de →
BD4A	4A		DECA		1←FE
BD4B	C6	07	LDB	##07	
BD4D	8D	27	BSR	#BD76	Si W,5←X
BD4F	CC	200B	LDD	##200B	20←DF
BD52	8D	22	BSR	#BD76	Si A,9→X

BD54	86	08	LDA	##08	8=F7
BD56	8D	23	BSR	##D7B	SI 0,7→X
BD58	C6	05	LDB	##05	
BD5A	8C	0009	CMPX	##0009	
BD5D	26	02	BNE	##D61	SI ↑ ou A,1X
BD5F	30	18	LEAX	-8,X	Si → ou S,3→X
BD61	8D	13	BSR	##D76	
BD63	1F	10	TFR	X,D	
BD65	54		LSRB		1 ou 2 touches ?
BD66	24	01	BHS	##D69	2 touches
BD68	59		ROLB		1 ou 3 touches
BD69	C1	08	CMPB	##08	
BD6B	23	01	BLS	##D6E	
BD6D	5F		CLRB		3 ou 4 touches
BD6E	DD	57	STD	##57	
BD70	86	02	LDA	##02	Type entier
BD72	8E	6155	LDX	##6155	Adr.AC.flottant
BD75	39		RTS		Retour au BASIC
BD76	1A	01	ORCC	##01	1→C
BD78	79	E7C9	ROL	##E7C9	Ligne suivante (TO7)
BD7B	C0	02	SUBB	##02	
BD7D	B5	E7C8	BITA	##E7C8	Touche lue appuyée ?
BD80	26	02	BNE	##D84	Non appuyée
BD82	30	85	LEAX	B,X	B+X→X
BD84	39		RTS		

REMARQUE 1 : En cas d'appui simultané sur trois touches (ou sur les quatre), il y a retour de la valeur 0 ; il en est de même si aucune touche n'est appuyée.

REMARQUE 2 : Pour le TO7-70, il faut placer une valeur de 0 à 7 sur les bits 0, 1 et 2 de \$E7C9 pour sélectionner une ligne du clavier ; on devra donc modifier \$BD34 (valeur 07 au lieu de FE) et \$BD78 (DEC \$E7C9 au lieu de ROL \$E7C9, c'est-à-dire 7A au lieu de 79).

3. Utilisation

La routine précédente sera implantée dans un programme devant utiliser la lecture du clavier en écrivant au début de celui-ci les instructions suivantes :

```
10 CLEAR,&HBD00:DEFUSR0=&HBD30
20 POKE &H6295,98:POKE &H6296,57:POKE &H6294,50
30 FOR I=&HBD30 TO &HBFFF
40 READ A$:IF A$(<"FIN" THEN POKE I,VAL("&H"+A$):NEXT
2000 DATA BD,24,C3,86,....,30,85,39,FIN
```

REMARQUE 1 : à la ligne 2000, les adresses (\$24C3, etc...) devront bien sûr être écrites octet par octet.

D'autre part, l'adresse \$BD30 écrite ici est purement indicative.

REMARQUE 2 : si on désire économiser de la place en mémoire, par exemple pour le TO7 si on ne dispose pas de l'extension, on pourra mettre les DATA en tête du programme (lignes 10, 13 et 16) et implanter la routine directement dans les DATA eux-mêmes en écrivant :

```
30 DEFUSR0=&H65FA:FOR I=&H65FA TO &H7FFF
```

4. Autres possibilités

Les instructions précédentes permettront d'utiliser USR0(I) exactement comme STICK(I), les deux joueurs pouvant donc jouer simultanément, et indépendamment bien sûr !

On pourrait de même créer très facilement une routine appelée par USR1(I), lisant par exemple la touche "ENTREE" pour le joueur 0 et la touche "RAZ" pour le joueur 1 ; on pourrait ainsi remplacer exactement la fonction STRIG lisant l'état des boutons de manettes de jeu.

Pour cela, on enverra la valeur &HFD (06 pour le TO7-70) en \$E7C9 ; selon la valeur de l'argument du sous-programme, on lira ensuite le bit 3 (touche RAZ) ou le bit 4 (touche ENTREE) de \$E7C8.

2

Recettes classiques d'amélioration des performances

En suivant les quelques conseils donnés ici, on pourra "sans effort" accroître sensiblement la vitesse d'exécution d'un programme, en général d'environ 10 à 15 %.

1. Écrire en début de programme les variables les plus souvent utilisées, en leur affectant une valeur (et non pas en les écrivant dans un calcul : voir 2^e partie) ; elles seront ainsi créées au début de la zone des variables, et donc trouvées plus vite à chaque utilisation.
2. Utiliser des variables entières (définition DEFINT) chaque fois que cela est possible ; les calculs sont alors beaucoup plus simples, et donc plus rapides.
3. Utiliser l'instruction INC de la 3^e partie pour tous les calculs du type $X=X+\text{expression}$.
4. Remplacer les constantes numériques souvent utilisées, et comportant 3 chiffres ou plus, par une variable (CT1, CT2, etc...) ; ceci évite de répéter le calcul de la constante, relativement long puisqu'effectué à partir des codes ASCII de chaque chiffre.

5. Essayer d'utiliser des boucles FOR ou WHILE (avec la routine de la 3^e partie) au lieu de IF ou GOTO.

6. Utiliser ON au lieu de IF.

7. N'utiliser que des variables simples ou des tableaux à une seule dimension, ceci étant en principe toujours possible ; la recherche d'un élément est alors nettement plus rapide.

Un tableau à deux dimensions $A(n,m)$ sera donc par exemple remplacé par le tableau $B((n+1)*(m+1)-1)$, dont tous les éléments $B(K)$ pourront être atteints "ligne par ligne" par :

```
FOR I=0 TO n*(m+1) STEP m:FOR K=I TO I+m-1
```

8. Ne pas écrire dans la mesure du possible la variable de contrôle d'une boucle FOR après le NEXT ; ceci permet en effet d'éviter la recherche de la variable à chaque passage par le NEXT.

On notera enfin qu'optimiser la vitesse d'exécution d'un programme conduira souvent à augmenter son occupation mémoire, et inversement !...

Compilation interactive des adresses de variables et des constantes

Lorsque l'interpréteur rencontre une variable dans le programme, il recherche celle-ci dans la zone correspondante, située juste après le programme lui-même ; s'il ne l'y trouve pas, la variable est ajoutée à la fin de la zone, après déplacement de la zone des tableaux (ceci dans le cas d'une affectation ou d'un FOR, etc...).

On peut en déduire qu'une variable créée en mémoire n'est jamais déplacée lors de l'exécution, quelque soit son type (numérique ou chaîne).

A partir de cette observation, nous allons voir qu'il est possible de "court-circuiter" la routine \$A48 de recherche d'une variable ; on obtiendra ainsi une amélioration des performances beaucoup plus nette que celle que l'on peut obtenir par les seuls moyens précédents.

I. La méthode

1. Principe

Lorsque l'interpréteur rencontre un nom de variable, nous remplaçons dans le programme lui-même le nom de la variable par son adresse ; celle-ci est donc compilée automatiquement lors de la première exécution de l'instruction correspondante.

Lorsque l'interpréteur repassera ensuite sur la même instruction (cas d'une boucle), il pourra alors aller directement à la valeur correspondante, sans aucune recherche.

2. Précautions à observer

Une adresse étant codée sur 2 octets, la compilation ne pourra être effectuée que si le nom de la variable comporte au moins deux caractères, ou un caractère suivi d'un espace ; déplacer la fin du programme pour ajouter un espace déplacerait en effet les variables, rendant fausses les adresses déjà compilées.

Nous avons vu d'autre part qu'on ne pouvait pas placer dans le programme lui-même les valeurs &H0,22,3A,89 et FF (à cause de la routine \$66B cherchant la fin d'une instruction, et utilisée par GOTO,IF,FOR, etc...), ou &H8F81,8FAF,C481, etc... (routine \$16AD utilisée par FOR ET WHILE).

Il est évident qu'on ne peut pas non plus écrire comme premier octet d'une adresse une valeur correspondant à un code d'instruction ; par exemple après un THEN, une adresse \$80xx serait interprétée comme étant une instruction THEN END, d'où l'arrêt du programme !

Enfin, l'interpréteur devra pouvoir distinguer une adresse (déjà compilée) d'un nom de variable non encore compilé, qui commence toujours par un octet contenant une valeur de &H41(A) à &H5A(Z) (les noms de variable sont en effet toujours codés en majuscules, même si on les écrit en minuscules) ; une adresse ne devra pas non plus être confondue avec une constante, ou une parenthèse, etc...

Finalement, le premier octet d'une adresse compilée ne pourra être compris qu'entre &H5D et &H7F ; on utilisera en effet les valeurs inférieures à &H20 pour la compilation des constantes.

Le deuxième octet devra être différent de &H0,22,3A,89 et FF.

Si une de ces deux conditions n'est pas respectée, la variable ne sera simplement pas compilée, d'où un fonctionnement "normal".

REMARQUE: La première condition est toujours vérifiée pour le TO7 sans extension mémoire; ceci permet dans ce cas de simplifier la routine en évitant le problème du déplacement évoqué ci-après.

3. Mise en œuvre

Toutes les variables d'un programme ont une adresse supérieure ou égale au contenu α de \$611E (début de la zone des variables: voir 2^e partie).

Au lieu d'implanter l'adresse réelle γ d'une variable, on implantera donc la valeur $\gamma+d$, d étant un déplacement (négatif ici) égal à $\#\$5B01-\alpha$.

La première variable aura donc son adresse compilée par la valeur &H5B01, etc...; bien entendu, une variable sera ensuite retrouvée en retranchant le déplacement d à l'adresse.

Ceci permettra de compiler un nombre maximal de variables différentes, qui pourront donc occuper 9471 octets (valeurs de &H5B01 à &H7FFE, &H7FFF n'étant pas compilée à cause du FF); ceci correspond à environ 1200 variables réelles différentes; ce nombre ne sera à coup sûr jamais atteint, quelque soit l'application, et la taille du programme!

Cette constatation permet de se passer en pratique du test: "adresse compilée \leq &H8000", et ceci même dans le cas du TO7-70.

Le déplacement d sera calculé une fois pour toutes en début de programme par la routine d'initialisation de la compilation.

4. Initialisation

Notre routine, à écrire, devra remplacer la routine \$A48 de recherche d'une variable; cette dernière commence en \$A4D par une instruction JSR \$6297, adresse où l'on trouve RTS suivi de 2 octets inemployés (voir 3^e partie).

D'autre part, notre compilation va modifier comme on le verra le traitement des opérandes, c'est-à-dire la routine \$770 commençant par JSR \$627C.

Le sous-programme d'initialisation doit donc calculer le déplacement d, rangé ici en \$BFFB, et placer en \$6297 et \$627C des instructions de déroutement vers les nouvelles routines.

Ces dernières ne deviendront donc actives qu'après exécution de ce sous-programme d'initialisation (par EXEC), dont le listing est le suivant :

BD01	CC	BE31	LDD	##BE31	Trait. des opérandes
BD04	FD	627D	STD	\$627D	
BD07	C6	97	LDB	##97	Recherche d'1 var.
BD09	FD	6298	STD	\$6298	
BD0C	86	7E	LDA	##7E	Code de JMP
BD0E	B7	627C	STA	\$627C	
BD11	B7	6297	STA	\$6297	
BD14	CC	5B01	LDD	##5B01	Première adresse
BD17	93	1E	SUBD	\$1E	Début zone des var.
BD19	FD	BFFB	STD	\$BFFB	Déplacement des var.
BD1C	39		RTS		

II. Compilation d'une adresse

1. Action de la routine

Notre routine remplace dans le programme lui-même le nom de la variable cherchée par l'adresse $\gamma = \gamma + d$ (si cette adresse est compilable), γ étant l'adresse dans la zone des variables du premier octet précédant le nom ; cet octet contient en effet la type de la variable et le nombre de caractères du nom diminué de 1 (voir 2^e partie).

Lorsque la routine est appelée pour une adresse γ' déjà compilée (lors d'une exécution antérieure de la même instruction), il y a calcul de l'adresse $\gamma = \gamma' - d$ pointant sur le premier octet ; d'où la détermination du type, envoyé en \$6105, et de l'adresse du premier octet de la valeur, placée en \$613D et dans le registre X ; on constate donc qu'il n'y a plus aucune recherche de la variable, d'où bien sûr accroissement de la rapidité.

Il faut noter que notre routine crée toujours les variables non encore existantes en mémoire, même écrites dans un calcul d'expression (ceci car on n'a alors plus la valeur &H803 située au sommet de la pile, mais seulement "un cran en dessous" : voir \$AA2) ; ceci pourrait provoquer une erreur dans

le cas de l'affectation de la valeur d'une expression à un élément de tableau; ce cas est donc détecté, et l'adresse de l'élément de tableau (rangée en \$613F par le début du traitement de l'affectation) corrigée alors en conséquence.

Enfin, il ne peut y avoir compilation en cas d'utilisation d'un élément de tableau, l'indice pouvant être une variable, prenant donc différentes valeurs correspondant à des adresses différentes.

2. La routine

Elle utilise un sous-programme effectuant la compilation, placé avant la routine proprement dite; le registre X contenant une valeur γ et l'accumulateur D un déplacement d , ce sous-programme place si cela est possible (au moins deux octets disponibles, et pas de &H0,22, etc... dans la valeur à ranger) la valeur $\gamma+d$ à l'adresse α contenue dans le registre Y; les octets disponibles éventuels (si le nom de la variable comporte plus de deux caractères) situés en $\alpha+2$, $\alpha+3$, etc..., sont remplacés par le caractère "espace" (celui-ci étant ignoré par l'interpréteur: voir \$61B8).

Le sous-programme est le suivant:

BE6C	31	22	LEAY	2,Y	Au moins 2 octets
BE6E	109C	B9	CMPLY	#B9	Y sont-ils ?
BE71	22	1D	BHI	#BE90	Non \Rightarrow pas de compil.
BE73	33	8B	LEAU	D,X	$\gamma' = \gamma + d \rightarrow U$
BE75	1F	30	TFR	U,D	$\gamma' \rightarrow A$ et B
BE77	5C		INCB		A-t-on FF ?
BE78	27	16	BEQ	#BE90	Oui \Rightarrow pas de compil.
BE7A	5A		DECB		A-t-on 0 ?
BE7B	27	13	BEQ	#BE90	Oui \Rightarrow pas de compil.
BE7D	C1	22	CMPB	#22	
BE7F	27	0F	BEQ	#BE90	Pas de compilation
BE81	C1	3A	CMPB	#3A	
BE83	27	0B	BEQ	#BE90	Pas de compilation
BE85	C1	89	CMPB	#89	
BE87	27	07	BEQ	#BE90	Pas de compilation
BE89	EF	3E	STU	-2,Y	Rangement γ'
BE8B	109C	B9	CMPLY	#B9	Plus de 2 caract. ?
BE8E	26	01	BNE	#BE91	Oui \Rightarrow les effacer
BE90	39		RTS		
BE91	C6	20	LDB	#20	Code de "espace"
BE93	E7	A0	STB	,Y+	
BE95	20	F4	BRA	#BE8B	Boucle

La routine proprement dite est la suivante :

BE97	109E	B9	LDY	\$B9	Adr.1°caract.nom
BE9A	32	62	LEAS	2,S	Dépile retour \$6297
BE9C	81	5A	CMPA	##5A	Code de "Z"
BE9E	22	3A	BHI	\$BEDA	Adr.déjà compilée
BEA0	8D	0A0A	JSR	\$0A0A	Lit nom de la var.
BEA3	81	25	CMPA	##25	A-t-on I, #, \$ ou % ?
BEA5	22	04	BHI	##EAB	Non
BEA7	9D	B2	JSR	\$B2	Caract.suivant→A
BEA9	9F	B9	STX	\$B9	Replace après nom
BEAB	1F	89	TFR	A,B	
BEAD	9D	B8	JSR	\$B8	
BEAF	C1	28	CMPB	##28	Code de "("
BEB1	26	03	BNE	\$BEB6	Variable simple
BEB3	7E	0A52	JMP	\$0A52	Élément de tableau
BEB6	DE	20	LDU	\$20	Début zone tableaux
BEB8	1193	3F	CMPU	\$3F	
BEBB	23	05	BLS	\$BEC2	Affectation à 1 tabl.
BEBD	8D	0A52	JSR	\$0A52	Aucun tableau
BEC0	20	00	BRA	\$BECF	Compilation
BEC2	34	40	PSHS	I	Contenu \$20→pile
BEC4	8D	0A52	JSR	\$0A52	Traite variable
BEC7	DC	20	LDD	\$20	A peut être bougé
BEC9	A3	E1	SUBD	,S++	Ancien \$20
BECB	D3	3F	ADDD	\$3F	
BECD	DD	3F	STD	\$3F	Adr.elt.tab.corrigée
BECF	FC	BFFB	LDD	\$BFFB	Valeur déplacement
BED2	C	3C	INC	\$3C	Nbr.caract.du nom
BED4	D0	3C	SUBB	\$3C	
BED6	82	00	SBCA	##00	Cas d'une retenus
BED8	20	92	BRA	\$BE6C	Compilation et RTS
BEDA	AE	A1	LDX	,Y++	Adr.y compilée→X
BEDC	109F	B9	STY	\$B9	Après adresse
BEDF	FC	BFFB	LDD	\$BFFB	Valeur déplacement
BEE2	43		COMA		Changement signe
BEE3	50		NEGB		
BEE4	26	01	BNE	\$BEE7	
BEE6	4C		INCA		—d→D
BEE7	30	8B	LEAX	D,X	Y—d→X
BEE9	E6	81	LDB	,X++	1° octet var.→B
BEEB	1F	98	TFR	B,A	... et→A
BEED	44		LSRA		
BEED	44		LSRA		
BEED	44		LSRA		
BEED	44		LSRA		Type variable
BEF1	97	05	STA	\$05	
BEF3	C4	0F	ANDB	##0F	Nbr.caract.—1
BEF5	30	85	LEAX	B,X	Adresse valeur→X
BEF7	9D	B8	JSR	\$B8	Saute les espaces
BEF9	9F	3D	STX	\$3D	
BEFB	39		RTS		

REMARQUE: Si on compilait l'adresse (déplacée toujours) du premier octet de la valeur, et non celle de l'octet précédant le nom, il faudrait ajouter un 3^e octet pour le type de la variable.

Ceci simplifierait la routine, mais obligerait à prévoir lors de l'écriture du programme BASIC à compiler 3 octets pour chaque variable, au lieu de 2 seulement, d'où une contrainte que nous avons préféré éviter.

III. Modification du traitement des opérandes

1. Le principe

On a vu dans la deuxième partie que les opérandes d'une expression sont traités par la routine \$77Ø; il faut donc obligatoirement modifier cette dernière pour que la rencontre d'une adresse déjà compilée (1^{er} octet entre &H5B et 7F) provoque le branchement en \$8ØØ, où il y aura appel de \$A48 et donc de notre routine de traitement d'une adresse.

La routine donnée ci-après compile aussi les constantes entières, en remplaçant la valeur initiale (codée en ASCII chiffre par chiffre) par la valeur binaire, déplacée de &H1Ø1, codée sur 2 octets.

Le déplacement de &H1Ø1 permet de compiler toutes les constantes comprises entre &HØ et &H1EFD, c'est-à-dire toutes les valeurs inférieures ou égales à 7933; le premier octet doit en effet être inférieur à &H2Ø, sous peine d'être confondu avec un espace et donc ignoré.

Pour les constantes entières négatives, c'est la valeur absolue qui est compilée, d'où encore un premier octet compris entre &HØ1 (Ø ne serait pas compilé) et 1F.

Enfin, une constante ne sera compilée que si elle occupe deux caractères au moins, ou bien sûr un caractère seulement suivi d'un espace (dans certains cas, l'espace peut aussi être situé avant le chiffre).

2. La routine

Elle traite les constantes numériques à compiler (1^{er} octet compris entre &H3Ø et &H39) ou déjà compilées (1^{er} octet inférieur à &H2Ø), et branche en \$8ØØ ou en \$785 selon que l'on a une variable (déjà compilée ou non), ou bien une fonction ou un caractère spécial (&.,',—,NOT, etc...).

Le listing est le suivant :

<u>BE31</u>	32	62	LEAS	2,S	Dépile retour \$627C
<u>BE33</u>	9E	B9	LDX	#B9	Avant 1 ^{er} caract.
<u>BE35</u>	9D	B2	JSR	#B2	Code 1 ^{er} caract.→A
<u>BE37</u>	31	01	LEAY	1,X	Sur 1 ^{er} caract.
<u>BE39</u>	24	14	BHS	#BE4F	Pas chiffre
<u>BE3B</u>	BD	077B	JSR	#077B	Valeur Cte→AC flot.
<u>BE3E</u>	9D	CD	JSR	\$CD	Type constante ?
<u>BE40</u>	2A	07	BPL	#BE49	≠2⇒pas de compil.
<u>BE42</u>	9E	57	LDX	#57	Valeur→X
<u>BE44</u>	8C	1EFD	CMPX	#1EFD	
<u>BE47</u>	23	01	BLS	#BE4A	Valeur compilable
<u>BE49</u>	39		RTS		
<u>BE4A</u>	CC	0101	LDD	#0101	Valeur déplacement
<u>BE4D</u>	20	1D	BRA	#BE6C	Compilation;RTS
<u>BE4F</u>	81	1F	CMPA	#1F	
<u>BE51</u>	22	0B	BHI	#BE5E	
<u>BE53</u>	EC	A1	LDD	,Y++	Cte déjà compilée
<u>BE55</u>	83	0101	SUBD	#0101	Correction
<u>BE58</u>	109F	B9	STY	\$B9	Positionne après Cte
<u>BE5B</u>	7E	0C59	JMP	#0C59	D→\$57;#2→\$05;RTS
<u>BE5E</u>	81	41	CMPA	#41	Code de "A"
<u>BE60</u>	25	07	BLO	#BE69	Caractère spécial
<u>BE62</u>	81	80	CMPA	#80	
<u>BE64</u>	24	03	BHS	#BE69	Fonction ou -,NOT..
<u>BE66</u>	7E	0800	JMP	#0800	Var.(compil,ou non)
<u>BE69</u>	7E	0785	JMP	#0785	Autre

REMARQUE: Le branchement au sous-programme de compilation suppose que ce dernier est placé juste après la routine.

IV. Mise en œuvre-résultats

1. Implantation-utilisation

Le sous-programme d'initialisation et les deux routines précédentes seront implantées en mémoire par le programme de création de la troisième partie.

Tous les branchements pointant dans les routines elles-mêmes étant relatifs, on pourra bien sûr comme toujours choisir une adresse d'implantation quelconque ; si on prend par exemple \$BE31, on ajoutera au programme de création les lignes suivantes :

```
1800 DATA Init.,&HBD01,CC,BE,31,...,39,FIN
2800 DATA Trait. opérandes,&HBE31,32,...,20,F4,FIN
2900 DATA Rech. Variable,&HBE97,109E,...,39,FIN
```

On trouvera au dernier chapitre la liste complète des DATA, avec le résultat de l'exécution du programme de création.

Bien entendu, on enregistrera après exécution les routines sur bande.

On pourra ensuite ajouter en tête de n'importe quel programme qui tourne :

```
10 CLEAR,&HBD00:LOADM:EXEC &HBD01
```

Toutes les constantes et variables du programme comportant au moins deux caractères, ou un seul suivi d'un espace, seront alors automatiquement compilées au fur et à mesure de l'exécution, ceci quelque soit la manière dont elles sont employées (par exemple dans un calcul, ou une instruction comme LOCATE,BOX,IF,etc..., ou comme indice de tableau, etc...).

Le résultat sera, selon les programmes, un accroissement de l'ordre de 60 % à 80 % de la rapidité d'exécution ; l'accroissement sera d'ailleurs d'autant plus grand que le programme utilise un nombre plus élevé de variables différentes, c'est-à-dire que le gain croît avec la complexité du programme.

2. Exemple

Soit le petit programme suivant :

```
10 CLEAR,&HBD00:EXEC &HBD01
20 A =11:BBBB=A +57:A=A +A
30 FOR I=&H660B TO &H6626
40 PRINT USING"% %"HEX$(PEEK(I)):NEXT
50 PRINT:PRINT "A,BBBB ="A,BBBB
```

Après lecture en mode direct des routines, l'exécution permet d'obtenir le contenu de la ligne 20 après compilation; on a :

66	27	0	14	5B	1	D4	1	C	3A	5B	7	20	20
D4	5B	1	C7	35	37	3A	41	D4	5B	1	C7	41	0

A,BBBB = 22 68

On constate que la variable A est compilée (adresse déplacée égale à \$5B01 comme prévu) seulement si elle est suivie d'un espace; les deux derniers caractères de la variable BBBB sont remplacés par des espaces (code &H20).

La constante 11 est compilée (valeur &H101+11=&H10C), mais pas la constante 57; elle devrait en effet être remplacée par la valeur &H13A, rejetée à cause du &H3A dans le second octet.

3. Quelques remarques

1. Après exécution du sous-programme d'initialisation, nos routines resteront bien sûr actives, même après un NEW éventuel; pour revenir à un fonctionnement normal, il faudra faire simplement :

POKE &H6297,&H39:POKE &H627C,&H39

La touche "initialisation" ne provoque en effet pas la réinitialisation de la RAM, effectuée seulement à la mise sous tension du T07.

2. Un programme compilé ne pourra être redémarré après l'arrêt normal que par GOTO α , α étant le numéro de la première ligne; un RUN "effacerait" (en fait, RUN réinitialise entre autres les pointeurs contenus en \$6120 et \$6122 avec l'adresse de fin du programme) en effet la zone des variables, qui ne pourraient pas être recrées à partir des adresses.

Pour la même raison, on ne peut pas modifier un programme déjà compilé.

3. Il est parfaitement possible de combiner nos routines de compilation avec les nouvelles fonctions et instructions de la 3^e partie.

Dans le cas des procédures, et pour des raisons évidentes, la récursivité ne marchera cependant plus; il suffira pour l'autoriser quand même de faire un simple *POKE&H6297,&H39* dans le programme principal avant l'appel par

CALL de la procédure récursive; on écrira bien sûr *POKE &H6297,&H7E* juste après le même CALL pour reprendre la compilation au retour de la procédure.

On notera que ceci permet de compiler quand même les constantes (et les branchements: voir chapitre suivant) dans la procédure elle-même, d'où un certain accroissement de la vitesse d'exécution de cette dernière.

4. Enfin, il ne faudra pas s'étonner d'obtenir des résultats bizarres en listant un programme compilé !...

V. Le problème des tableaux

Il est à priori possible, et souhaitable, de pouvoir compiler non plus seulement les adresses de variables, mais aussi celles des tableaux éventuels utilisés par un programme.

Malheureusement, on a vu que toute la zone des tableaux est déplacée à chaque création d'une nouvelle variable, problème n'existant pas avec ces dernières.

On a vu de plus qu'il n'est pas possible de compiler les adresses des éléments d'un tableau, au moins dans le cas où un des indices est une variable.

On ne peut donc finalement compiler que l'adresse du *nom* d'un tableau, et uniquement dans le cas où toutes les variables auront été créées avant la compilation de cette adresse.

On constate donc que ceci constitue une contrainte à respecter lors de l'écriture du programme (remarquons quand même que cette contrainte existe dans tous les langages structurés...) pour un bénéfice relativement réduit, ceci d'autant plus que le nombre de tableaux différents utilisés par un programme est en général petit, d'où une recherche dans la zone des tableaux très rapide de toute façon.

Il est cependant possible de modifier les routines précédentes pour traiter aussi les tableaux; en particulier, il faudra appeler la routine \$A52 avec la valeur 1 placée en \$6107 (il y a alors recherche du seul nom du tableau), le cas de DIM étant à traiter à part (on a alors \$6104 différent de 0: voir \$A01) pour permettre la création du tableau.

On calculera ensuite les valeurs des indices, que l'on empilera dans la pile S; il faudra de plus ranger dans la 2^e pile U (créée par exemple en \$BFFA) l'adresse de l'en-tête et pour chaque indice le contenu des mémoires \$6104 et \$6105, ceci pour le cas où un des indices est lui-même élément d'un autre tableau, chose à priori possible !

L'adresse de l'élément sera ensuite trouvée par un JMP \$BE9 (voir 2^e partie).

4

Compilation des adresses de branchement

Chaque fois que l'interpréteur rencontre un branchement (GOTO, GOSUB, ON), il y a calcul de l'étiquette (routine \$6FD : voir 2^a partie), puis recherche dans le programme lui-même de la ligne correspondante (routine \$4A0).

Nous présentons une méthode permettant de remplacer automatiquement l'étiquette de branchement par l'adresse effective (déplacée) du début de la ligne correspondante.

Le résultat est bien sûr un nouvel accroissement de la vitesse d'exécution d'un programme.

1. La méthode

1. Principe

Il est ici moins immédiat d'intervenir sur les routines de traitement des branchements, puisque celles-ci ne comportent aucun passage par la RAM

(à part la routine \$61B2, non modifiable en pratique car on ralentirait forcément le traitement de tous les caractères du programme, d'où un résultat inverse de celui recherché !).

On est donc obligé d'intervenir au niveau du décodage des instructions d'un programme.

On a vu en effet qu'à chaque code d'instruction rencontré par l'interpréteur, il y a passage en \$6270 (voir routine \$2B25), où l'on pourra dérouter le BASIC vers une routine testant si l'on a un code correspondant à GO(&H87) ou à ON(&H96), que l'on pourra alors traiter.

2. Précautions à observer

Les adresses effectives de branchement peuvent être à priori comprises en \$65F4 (adresse du \$ précédant la 1^{re} ligne du programme) et \$BFFF (valeur jamais atteinte en fait, un programme comportant toujours des variables !) pour le TO7, ou \$DFFF (même remarque) pour le TO7-70.

Il est parfaitement possible ici de placer dans le programme une valeur supérieure ou égale à &H80, l'interpréteur attendant en effet toujours une étiquette après un GOTO ou GOSUB ; il n'y a donc pas de risque de confusion avec une instruction.

Par contre, on ne peut toujours pas placer les valeurs &H0,22,3A,89 ou FF dans le programme, ainsi que les valeurs &H8F ou C4 suivies de &H81,82,AF ou B0.

γ étant l'adresse effective du branchement à la ligne k, c'est-à-dire l'adresse du 0 précédant la ligne, on implantera donc en fait l'adresse $\gamma' = \gamma + \&H2A0D$.

Dans le cas du TO7 avec extension mémoire, on obtient en effet ainsi un premier octet compris entre &H90 et E9 ; il faudra donc tester seulement si on a &HC4 (l'adresse effective commence alors par &H9A), auquel cas on ne compilera pas l'adresse.

Comme au chapitre précédent, il n'y aura pas non plus compilation si le 2^e octet est égal à &H0,22,3A,89 ou FF, ou si l'étiquette k ne comporte qu'un chiffre non suivi d'un espace.

La compilation elle-même sera donc toujours effectuée par le sous-programme du chapitre précédent.

REMARQUE: *cas du T07-70*:

Le déplacement de $\&H2A0D$ permet de compiler toutes les adresses de branchement inférieures ou égales à $\$D4F1$ (au delà, on obtient un $\&HFF$ non compilable), ce qui correspond à un programme d'environ *28K-octets*.

Si l'on désire compiler des programmes encore plus importants, il suffira de prendre un déplacement de $\&HD50D$, qui correspond à *soustraire $\&H2AF3$* de l'adresse; on obtient des adresses déplacées démarrant à $\$3B01$, d'où la compilation possible de programmes de plus de 49K-octets.

On devra alors bien sûr tester si le premier octet est égal à $\&H89$ ou $8F$ ou $C4$, auquel cas on ne compilera pas l'adresse.

3. Initialisation

Il suffit d'ajouter dans le sous-programme d'initialisation du chapitre précédent les instructions implantant un $JMP \$BDB8$ en $\$6270$.

L'adresse $\$BDB8$ peut bien sûr être modifiée à volonté; nous l'avons choisie pour que les routines données ci-après soient placées juste avant la routine de traitement des opérandes, ceci à cause des branchements relatifs au sous-programme de compilation.

II. Traitement des instructions GOTO et GOSUB

Notre traitement est bien sûr calqué sur le traitement "normal", situé en $\$606$ (voir 2^e partie).

Lorsque l'interpréteur rencontre un GOTO ou GOSUB suivi d'un chiffre (C est alors positionné à 1 par $\$B2$), l'adresse effective est calculée par la routine $\$629$, puis déplacée de la valeur $\&H2A0D$ et rangée dans le programme.

Si on n'a pas un chiffre, c'est que l'étiquette k a déjà été compilée; on retranche alors simplement le déplacement.

La routine, implantée ici en \$BDF4, est la suivante :

BDF4	32	62	LEAS	2,S	Dépile retour \$6270
BDF6	9D	B2	JSR	\$B2	Octet après GO→A
BDF8	DE	B9	LDU	\$B9	Adr.car.courant→U
BDFA	81	BC	CMPA	##BC	Code de SUB
BDFC	26	09	BNE	##E07	Traite TO
BDFE	9E	2C	LDX	\$2C	N° ligne courante→X
BE00	34	52	PSHS	A,X,U	
BE02	8D	03	BSR	##E07	Branchement ligne k
BE04	7E	2AED	JMP	\$2AED	Boucle d'exécution
BE07	9D	B2	JSR	\$B2	1 ^{er} caract.étiquette
BE09	31	41	LEAY	1,U	Adresse→Y
BE0B	24	1C	BHS	##E29	C=0→déjà compilé
BE0D	BD	06FD	JSR	\$06FD	Calcul de k
BE10	DE	B9	LDU	\$B9	Adr.caract.après k
BE12	34	40	PSHS	U	Emplément
BE14	BD	0629	JSR	\$0629	Trouve adr.effective
BE17	35	40	PULS	U	
BE19	DF	B9	STU	\$B9	Restaure \$B9
BE1B	1F	10	TFR	X,D	Adr.ligne k→D
BE1D	81	9A	CMPA	##9A	Donnerait C4
BE1F	27	05	BEQ	##E26	Pas de compilation
BE21	CC	2A0D	LDD	##2A0D	Valeur déplacement
BE24	8D	46	BSR	##E6C	SP. de compilation
BE26	9F	B9	STX	\$B9	Car.cour.sur déb. k
BE28	39		RTS		
BE29	EC	A4	LDD	,Y	Adresse compilée→D
BE2B	B3	2A0D	SUBD	##2A0D	Correction
BE2E	DD	B9	STD	\$B9	Car.cour.sur déb.k
BE30	39		RTS		

REMARQUE: Le branchement au sous-programme de compilation suppose que la routine est placée juste avant celui-ci.

III. Traitement de l'instruction ON

Le traitement "normal" est effectué en \$36B5, où on ira en cas d'instruction ON ERROR ou ON PEN.

L'instruction "ON expression" suivie de GOTO ou GOSUB est par contre traitée par la routine suivante; on constatera qu'elle fait bien sûr appel au traitement de GOTO et GOSUB vu ci-dessus.

Le listing est le suivant :

<u>BDB8</u>	27	08	BEQ	\$BDC2	Si 3A en tête (...)
<u>BDBA</u>	81	87	CMPR	##87	Code de GO
<u>BDBC</u>	27	36	BEQ	\$BDF4	GOTO,GOSUB
<u>BDBE</u>	81	96	CMPR	##96	Code de ON
<u>BDC0</u>	27	81	BEQ	\$BDC3	
<u>BDC2</u>	39		RTS		Autre inst.⇒\$2B28
<u>BDC3</u>	32	62	LEAS	2,S	Dépile retour \$8270
<u>BDC5</u>	90	B2	JSR	\$B2	Caract. après ON
<u>BDC7</u>	81	B8	CMPR	##B8	Code de PEN
<u>BDC9</u>	27	04	BEQ	\$BDCF	
<u>BDCB</u>	81	98	CMPR	##98	Code de ERROR
<u>BDCD</u>	26	03	BNE	\$BDD2	On a une expression
<u>BDCF</u>	7E	36B5	JMP	\$36B5	Traitement "normal"
<u>BDD2</u>	80	0EB8	JSR	\$0EB8	Valeur exp.→\$58
<u>BDD5</u>	C6	87	LDB	##87	Code de GO
<u>BDD7</u>	90	D0	JSR	\$D0	A-t-on GO ?
<u>BDD9</u>	34	02	PSHS	A	Emplie TO ou SUB
<u>BDDB</u>	A	58	DEC	\$58	
<u>BDDD</u>	26	04	BNE	\$BDE3	Pas bonne adresse
<u>BDDF</u>	35	02	PULS	A	TO ou SUB
<u>BDE1</u>	20	15	BRA	\$BDF8	Effectue branchement
<u>BDE3</u>	90	B2	JSR	\$B2	1 ^{er} caract.étiquette
<u>BDE5</u>	25	06	BLO	\$BDED	Chiffre⇒pas compilé
<u>BDE7</u>	90	B2	JSR	\$B2	Adr. déjà compilée
<u>BDE9</u>	90	B2	JSR	\$B2	On saute l'adresse
<u>BDEB</u>	20	03	BRA	\$BDF0	
<u>BDED</u>	80	06FD	JSR	\$06FD	Saute l'étiquette
<u>BDF0</u>	26	E9	BNE	\$BDD8	Étiquette suivante
<u>BDF2</u>	35	82	PULS	A,PC	Instruction suivante

REMARQUE: On notera au début de la routine (adresse \$BDB8 correspondant à celle implantée en \$6271 lors de l'initialisation) le branchement au traitement de GOTO et GOSUB, ou bien à celui de ON.

On remarquera aussi qu'une étiquette n'est compilée que lorsqu'il y a effectivement branchement à la ligne correspondante.

Mise en œuvre – exemple

Toutes les routines précédentes seront toujours créées par le même programme de création, puis enregistrées sur bande par SAVEM.

On veillera à placer les unes à la suite des autres la routine traitant les branchements, puis celle traitant les opérandes (le sous-programme de compilation étant placé en tête de celle-ci) et enfin celle traitant les variables.

On trouvera à la fin de cette 4^e partie la liste de tous les DATA correspondants.

Après exécution du sous-programme d'initialisation, il y aura donc compilation automatique de toutes les adresses de branchement, au fur et à mesure de l'exécution.

On notera toutefois que seuls ON, GOTO et GOSUB sont traités par nos routines; une étiquette placée dans une instruction IF après un THEN ou un ELSE ne sera donc pas compilée; si on écrit par contre GOTO k au lieu de THEN k, et ELSE GOTO k au lieu de ELSE k, il y aura bien compilation.

Exemple :

```
10 CLEAR,&HBD00:LOADM:EXEC &HBD01
20 GOSUB 25
23 ON AA GOTO 50,30,50
26 AA=2:RETURN
30 FOR I=&H660E TO &H6637
40 PRINTUSING"%  %",HEX$(PEEK(I)):NEXT
```

Ce programme, dont on ne s'inspirera pas vu l'imbrication des branchements..., écrit le contenu des lignes 20 à 26 après leur exécution.

On obtient:

66	18	0	14	87	BC	90	39	20	0	66	2D	0	17
96	20	5B	1	20	87	8B	20	35	30	2C	90	44	2C
35	30	0	66	38	0	1A	5B	1	D4	32	3A	8B	0

On constate à la ligne 20 que l'étiquette 26 est remplacée par la valeur &H9039=&H662C+&H2A0D, \$662C étant en effet l'adresse du 0 précédant la ligne 26.

A la ligne 23, l'étiquette 30 est bien compilée par la valeur &H9044=&H6637+&H2A0D; les deux étiquettes 50 ne sont par contre pas compilées puisqu'elles ne sont pas utilisées.

La variable AA est bien sûr compilée, ainsi que la variable I de la ligne 30.

5

Application à d'autres ordinateurs

Toutes les routines précédentes sont évidemment spécifiques à l'interpréteur des deux T07.

La méthode présentée, consistant en une compilation interactive de certaines adresses et constantes, est par contre très générale et peut être appliquée à n'importe quel ordinateur.

1. Cas où il existe des vecteurs en mémoire vive

Ce cas est heureusement le plus fréquent; il permet comme nous l'avons vu d'intervenir très facilement sur le fonctionnement de l'interpréteur.

Pour mettre notre méthode en œuvre, il faudra toujours étudier soigneusement les routines de l'interpréteur permettant "d'explorer" le programme.

On trouvera facilement l'emplacement de la routine cherchant la fin d'une instruction en observant le traitement des instructions "non exécutables" comme REM ou DATA, ou bien sûr en étudiant les instructions qui l'utilisent comme IF ou GOTO.

Cette routine existe sur tous les interpréteurs, et on trouvera très probablement dans le cas d'un ordinateur utilisant le code ASCII qu'elle traite les mêmes valeurs que pour les TO7, c'est-à-dire &H0 (fin de ligne), 22 (guillemets), 3A (fin d'instruction); les valeurs &H89 (IF, ceci pour le cas des IF imbriqués permis par les TO7) et FF (fonctions, codées sur 2 octets) ne seront peut être par contre pas traitées, ou seront différentes.

Rappelons que toutes les valeurs traitées par cette routine ne pourront pas être implantées dans un programme.

L'étude de l'instruction FOR (et WHILE lorsqu'elle existe) permettra ensuite de trouver la routine cherchant le NEXT (ou WEND) associé au FOR; on trouvera donc là aussi certaines valeurs à ne pas planter dans un programme.

On en déduira, en fonction de la carte mémoire de l'ordinateur considéré, les déplacements à effectuer pour compiler les constantes, les adresses de variables et celles des branchements; on devra d'ailleurs éventuellement ne compiler que les adresses conduisant à des valeurs comprises dans un certain intervalle, ce qui ne diminue pas l'intérêt de la méthode.

II. Cas où le BASIC est entièrement figé en ROM

Rappelons que les instructions de branchement des TO7 ne comportent aucun passage par la RAM, ce qui ne nous a pas empêché d'intervenir sur leur fonctionnement; il suffit en effet de décoder les instructions correspondantes, d'où un déroutement vers les nouvelles routines, écrites à partir de celles de l'interpréteur.

On fera donc de même s'il existe qu'un seul vecteur en RAM, situé dans la boucle d'exécution des programmes ou dans la routine de traitement des instructions.

Il ne sera pas beaucoup plus difficile d'intervenir sur un BASIC ne comportant même pas cela.

Il suffira en effet d'écrire alors sa propre boucle d'exécution des programmes (voir 2^e partie), toujours en "recopiant" celle de l'interpréteur.

En tête du programme BASIC à faire exécuter, on placera donc un simple EXEC (ou USR) vers la routine; toute l'exécution du programme sera alors

contrôlée par cette dernière, ce qui permet bien sûr toutes les interventions désirées.

On observera toutefois que le traitement de certaines instructions (GOSUB,NEXT,IF) se termine non pas par un simple RTS (provoquant le retour à la routine), mais par un saut à la boucle d'exécution des programmes "normale", d'où reprise du contrôle par l'interpréteur.

Il faudra donc faire traiter ces instructions par ses propres routines, à écrire encore en recopiant celles d'origine.

6

Résumé de la quatrième partie

Nous donnons ci-après la liste complète des DATA à écrire pour les TO7 et TO7-70 dans le programme de création de la troisième partie, en vue de permettre la compilation automatique des adresses utilisées par un programme BASIC.

Les routines correspondantes (ici de \$BD01 à \$BEFB) seront bien sûr enregistrées en binaire par SAVEM.

Un simple CLEAR, &HBD00:LOADM:EXEC &HBD01 ajouté en tête de n'importe quel programme aura alors pour résultat de pratiquement doubler la vitesse d'exécution de celui-ci.

Rappelons aussi que les DATA donnés ci-après peuvent être combinés avec ceux de la troisième partie ; on bénéficiera alors en même temps de toutes les instructions supplémentaires et de l'accroissement de la rapidité d'exécution.

Le listing des DATA est le suivant:

```

1695 '.....Initialisation.....
1800 DATA Initialisation,&HBD01,CC,BDB8,FD,6271,CC,BE31
,FD,627D,C6,97,FD,6298,86,7E,B7,6270,B7,627C,B7,6297,CC
,5B01,93,1E,FD,BFFB,39,FIN Init.
1990 '
1995 'Routines
2700 DATA Branchements,&HBD08,27,8,81,87,27,36,81,96,27
,1,39,32,62,9D,B2,81,B8,27,4,81,98,26,3,7E,36B5,BD,EB8,
C6,87,9D,D0,34,2,A,58,26,4,35,2,20,15,9D,B2,25,6,9D,B2,
9D,B2,20,3,8D,6FD,26,E9,35,82
2710 DATA 32,62,9D,B2,DE,B9,81,BC,26,9,9E,2C,34,52,8D,3
,7E,2AED
2720 DATA 9D,B2,31,41,24,1C,8D,6FD,DE,B9,34,40,8D,629,3
5,40,DF,B9,1F,10,81,9A,27,5,CC,2A0D,8D,46,9F,B9,39,EC,A
4,83,2A0D,D0,B9,39,FIN Branchements
2800 DATA Trait.operandes,&HBE31,32,62,9E,B9,9D,B2,31,1
,24,14,BD,77B,9D,CD,2A,7,9E,57,8C,1EFD,23,1,39,CC,101,2
0,1D,81,1F,22,8,EC,A1,83,101,109F,B9,7E,C59,81,41,25,7,
81,80,24,3,7E,800,7E,785
2810 DATA 31,22,109C,B9,22,1D,33,8B,1F,30,5C,27,16,5A,2
7,13,C1,22,27,F,C1,3A,27,B,C1,89,27,7,EF,3E,109C,B9,26,
1,39,C6,20,E7,A0,20,F4,FIN Sous-Prgm.
2900 DATA Rech.variables,&HBE97,109E,B9,32,62,81,5A,22,
3A,BD,A0A,81,25,22,4,9D,B2,9F,B9,1F,89,9D,B8,C1,28,26,3
,7E,A52,DE,20,1193,3F,23,5,BD,A52,20,D,34,40,8D,A52,DC,
20,A3,E1,D3,3F,D0,3F,FC,BFFB,C,3C,D0,3C,82,0,20,92
2910 DATA RE,A1,109F,B9,FC,BFFB,43,50,26,1,4C,30,8B,E6,
81,1F,98,44,44,44,44,97,5,C4,F,30,85,9D,B8,9F,3D,39,FIN
Rech.
2950 DATA FIN des routines

```

Résultat de l'exécution

ROUTINE Initialisation		
Debut:BD01	Fin:BD25	Somme: 5527
ROUTINE Branchements		
Debut:BD08	Fin:BE30	Somme: 12352
ROUTINE Trait.operandes		
Debut:BE31	Fin:BE96	Somme: 8628

ROUTINE Rech.variables

Debut:BE97 Fin:BEFB Somme: 10513

Implantation terminée...

Enregistrer de BD01 à BEFB (Par SAVEM)

REMARQUE: Pour le TO7 de base, les adresses seront bien sûr choisies à partir de \$7D01.

On pourra par contre choisir \$DD01 pour le TO7-70.

Enfin, la routine suivante, à implanter n'importe où (par exemple dans les DATA eux-mêmes), permet d'utiliser le clavier exactement comme les manettes de jeu: 2 joueurs jouant simultanément, déplacements en diagonale possibles (voir remarque du chapitre 1 pour le TO7-70: remplacer les 2 valeurs soulignées par 07 et 7A):

```
2000 DATA BD,24,C3/86,FE/B7,E7,C9/CC,2,B/8E,0,0,D,58,26  
      ,8,8D,32,8D,30,8D,2E,20,10,4A,C6,7,8D,27,CC,20,B,8D,22,  
      86,8,8D,23,C6,5,8C,0,9,26,2,30,18,8D,13,1F,10,54,24,1,5  
      9,C1,8,23,1,5F,0D,57,86,2,8E,61,55,39  
2010 DATA 1A,1,79,E7,C9,C0,2,B5,E7,C8,26,2,30,85,39,FIN
```


Cinquième partie

LE TO 9 LE DOS BASIC

Une des caractéristiques les plus visibles du dernier ordinateur TO9 de THOMSON est constituée par la présence d'un lecteur de disquettes ; nous étudions donc ici le DOS BASIC, c'est-à-dire l'extension de l'interpréteur permettant entre autres la gestion des disquettes ; nous présentons aussi un programme permettant l'exploitation optimale de ce DOS.

Le DOS des TO7-70 ou TO7 équipés d'un lecteur de disquettes est réalisé exactement de la même manière que celui du TO9 ; nous traitons donc aussi ce cas.

Le TO9 comporte plusieurs logiciels en mémoire morte ; nous indiquons ici comment accéder aux différentes ROM et comment les décoder, et cela tant pour le TO9 que pour le TO7-70 équipé du BASIC 128.

Enfin, la présence du DOS impose certaines modifications (mineures) de nos applications ; nous indiquons donc comment réaliser l'adaptation des routines correspondant aux nouvelles instructions de la troisième partie et à la compilation partielle des programmes BASIC de la quatrième partie.

1

LE TO 9 Ses logiciels

Le dernier micro-ordinateur de THOMSON représente un très grand pas en avant par rapport au TO7-70.

Tous les programmes BASIC écrits pour ce dernier pourront toutefois être traités par le TO9 puisque celui-ci contient le BASIC 1.0 en mémoire morte ; nous indiquons donc ici la marche à suivre pour cela.

Le TO9 contient aussi d'autres logiciels en ROM, dont le nouveau BASIC 128 ; nous verrons donc comment étudier ces logiciels, normalement « invisibles » depuis le BASIC 1.0.

Enfin, nous traitons le cas des possesseurs de TO7-70 équipés du BASIC 128, et désirant étudier celui-ci.

I. Présentation matérielle

1. Les progrès par rapport au TO7-70 sont évidents ; on notera donc en particulier le très bon clavier, très complet, la présence d'un lecteur de disquettes intégré de format 3,5 pouces et de bonne capacité (320 ko), le nouveau crayon optique très amélioré et la possibilité de connecter une « souris ».

D'autres progrès, bien que moins visibles, sont tout aussi importants, tels que l'affichage en 40 ou 80 colonnes, la palette des 16 couleurs à choisir parmi 4096, l'intégration de l'interface imprimante, les 128 ko de RAM en version de base, etc....

2. Ces différences matérielles, et en particulier celles liées à l'affichage et à la nouvelle gestion du clavier (bufférisé), ont imposé certaines modifications dans les registres utilisés par le moniteur :

- \$605F représente le mode d'affichage (40 ou 80 colonnes, ou « bit map ») ;
- \$6079 (deux octets) contient l'adresse du buffer clavier ;
- \$607B contient la taille de ce buffer, initialisée à 5 ;
- \$605E est le pointeur de lecture du buffer ;

les principaux registres, listés dans l'annexe 2, n'ont par contre pas été modifiés ; il en est de même pour les adresses des routines du moniteur (annexe 1).

II. Les logiciels ; leur étude

Le TO9 contient plusieurs logiciels en mémoire morte, dont le BASIC 1.0 des TO7 et TO7-70 et le nouveau BASIC 128.

Ces logiciels sont tous placés à la même adresse, soit de \$0000 à \$3FFF ; l'accès à l'une ou l'autre ROM est effectué par invocation d'une adresse comprise entre \$0000 et \$0003 :

- \$0000 permet l'accès aux 16 premiers ko octets du BASIC 128 ;
- les 16 ko suivants seront commutés par l'adressage de \$0001 ;
- \$0002 permet l'accès au BASIC 1.0 ;
- \$0003 commute une ROM contenant la deuxième partie du moniteur (la première étant située en \$E800).

Tous ces logiciels seront étudiés par le biais d'une recopie dans une zone de mémoire protégée (en \$D000), qui sera désassemblée avec correction automatique des adresses.

On ajoutera simplement pour cela les instructions suivantes au programme de désassemblage de la page 33 :

```
56000 'Désassemblage de toutes les ROM's(et RAM) TO9
56010 CLEAR,&HCFFF:CLS
56020 INPUT"ROM à désassembler(0 à 3,ou ENT) ";A$:ROM=V
AL(A$):IF ROM=0 AND A$<>"0" THEN ROM=2      'Basic 1.0
56030 FOR I=&HDF00 TO &HDF16
56040 READ A$:POKE I,VAL("&H"+A$):NEXT:POKE &HDF02,ROM
56050 DATA B7,0,0,8E,0,0,CE,D0,0,EC,81,ED,C1,11,83,DF,0
,25,F6,B7,0,2,39      'Commutation ROM;transfert en $D000
57430 IF ADR<&H4000 THEN POKE &HDF04,ADR @ 256:POKE &HD
F05,ADR MOD 256:DECAL=&HD000-ADR:ADRF=ADRF+DECAL:ADR=&H
D000:EXEC &HDF00 ELSE DECAL=0
57510 IND=0:COD2=0:PRINT#1,HEX$(ADR-DECAL);'Corrige adr
57870 A$=" "+HEX$(X-DECAL):GOTO58500 'Corrige branchts
```

On pourra donc désassembler ainsi le nouveau BASIC 128 ; ce dernier méritant une étude complète, nous nous contenterons de signaler qu'il est largement compatible avec le BASIC 1.0, tout en étant à la fois plus complet et plus efficace aux points de vue gestion de la mémoire et traitement des variables.

REMARQUE : Cas du TO7-70 avec BASIC 128

Les deux ROM du BASIC 128, commutées respectivement par invocation des adresses \$0000 et 0001 (ROM 0 et 1), seront désassemblées par exactement le même programme, dans lequel il suffira de modifier les trois lignes suivantes pour le rendre compatible avec le BASIC 128 :

```
56010 CLEAR 1000,&HCFFF:CLS
56020 INPUT"ROM à désassembler(0 à 1) ";A$:ROM=VAL(A$):
IF ROM<0 OR ROM>1 GOTO 56020
56050 DATA B7,0,0,8E,0,0,CE,D0,0,EC,81,ED,C1,11,83,DF,0
,25,F6,B7,0,0,39      'Commutation ROM;transfert en $D000
```

2

Le DOS Basic 1.0

La présence du DOS en mémoire modifie bien évidemment le traitement des programmes.

Le vocabulaire de base est en effet complété de 23 nouvelles instructions et de 12 nouvelles fonctions, ce qui modifie entre autres la manière dont est codé et traité un programme, ainsi que son implantation.

1. Implantation et codage d'un programme

1. A l'initialisation, le DOS est placé au début de la mémoire libre ; le programme de l'utilisateur est donc ensuite déplacé d'autant.

L'examen de la mémoire \$611C permet en effet d'obtenir l'adresse \$8E54.

Le programme 2 de la page 46 permet de vérifier qu'il s'agit bien là de l'adresse du début des programmes.

2. Si on écrit en tête de ce programme 2 les diverses nouvelles instructions ou fonctions, on constatera qu'elles sont codées de &HD6(DSKINI) à &HEC (DENSITY) pour les instructions, et de &HFFA7(DSKF) à &HFFB2(DSKI\$) pour les fonctions.

Il s'agit là des codes suivant immédiatement ceux déjà utilisés pour le BASIC de base.

II. Tables des nouvelles instructions et fonctions

1. Nous avons vu (page 49) que le codage des instructions d'un programme est réalisé à partir de la routine \$2A21, qui permet de "balayer" les tables des mots réservés.

Chaque mot tapé au clavier est en effet comparé à tous les mots de la première table, dont l'adresse (\$0092) est contenue en \$6202 ; lorsqu'on est arrivé à la fin (voir \$2A76), il y a retour en \$2A23 où le registre U est incrémenté de 10 ; il pointe donc sur \$620C où l'on trouvera l'adresse de la deuxième table des instructions ; on trouvera ensuite de même en \$6216 et 6220 les adresses des troisièmes et quatrièmes tables éventuelles, signalées par une valeur non nulle dans l'octet d'adresse U-1 (valeur égale au nombre de mots contenus dans la table).

Toutes les tables des instructions ayant été balayées, le registre U est repositionné (en \$2A5F) sur \$61FC ; il y a alors retour en \$2A21 ; on trouvera donc en \$61FC + 10, soit en \$6107, l'adresse de la première table des fonctions ; celle de la suivante sera ensuite trouvée 10 octets plus loin, en \$6211, puis celle de la troisième éventuelle en \$621B, etc....

Le programme 2 de la page 46 permet de lister toutes les informations relatives aux tables ; on obtient :

6201	56	0	92	2	6B	27	1	CF	0	20
620B	17	6A	E9	6B	CE	C	6B	85	6B	E5
6215	0	0	0	7	F3	0	0	0	7	F3

On constate qu'il existe deux tables supplémentaires par rapport au BASIC de base, situées en \$6AE9 et 6B85 ; elles contiennent respectivement les 23+12 nouvelles instructions et fonctions.

2. Le programme 3 de la page 48 (modifier bien sûr les adresses de la ligne 220) permet alors de lister tous les nouveaux mots du BASIC DOS, que l'on trouvera ci-après.

3. REMARQUE : DOS T07-70

On trouvera de même en \$620B le nombre des nouvelles instructions, suivi de l'adresse de la table des mots correspondants ; l'adresse où sont décodés et traités ces nouveaux mots sera trouvée en \$620E (voir ci-après).

Les informations concernant les nouvelles fonctions seront de même trouvées à partir de \$6210.

III. Exécution des nouvelles instructions et fonctions ; Tables des adresses

1. Nous avons vu (page 53, en \$2B4A) qu'un code d'instruction supérieur à &HD5 provoque le branchement à l'adresse contenue en \$620E, soit \$6BCE; on y trouve les instructions suivantes :

6BCE	CMPA	#\$EC	Code>EC (DENSITY)2
	BHI	\$6BDA	Oui=>pas inst. du DOS
	LDX	#\$6B57	Non=>adr. table adrs. insts. DOS->X
	SUBA	#\$D6	DSKINI->0.etc...
	JMP	\$2B33	Branche au traitement de l'instr.
6BDA	CMPA	#\$EC	
	BHI	\$6BE1	
	JMP	\$7F3	Syntax error
6BE1	JMP	[\$6218]	Contient \$7F3(SN Error)

La table des adresses des nouvelles instructions du DOS est donc située en \$6B57, c'est-à-dire juste après la table des nouveaux noms d'instructions.

2. Nous avons vu (page 65, en \$2AA1) qu'un code de fonction supérieur à &HFFA6 provoque le branchement à l'adresse contenue en \$6213, soit en \$6BE5 ; on y trouve :

```

$BE5  CMPB  #$64      Code>FFE2?
      BLS   $6BED     Non=>fonction du DOS
      JMP   [ $621D]   Oui=>7F3(SN Error)
6BED  SUBB  #$4E      DSKF->0...DSKI$->$16
      CMPB  #$12      STRING$ ou DSKI$?
      BHI   $6BFA     Oui=>plusieurs paramètres
      PSHS  B
      JSR   $7DE      Calcul du paramètre(unique)
      PULS  B
6BFA  LDX   #$6BB6    Adr.table adrs.fonctions DOS->X
      JMP   $2AE7     Branche au traitement

```

La table des adresses des nouvelles fonctions du DOS est donc située en \$6BB6, soit juste après celle des noms.

3. Les nouvelles instructions du DOS sont donc les suivantes :

Nom	code hexa	Adresse hexa	Nom	Code hexa	Adresse hexa
BACKUP	E5	82C7	KILL	D8	6EAB
CIRCLE	E7	77F0	LSET	DE	85FD
COPY	E6	842B	NAME	D9	6F3A
DENSITY	EC	8597	PAINT	E8	789E
DEVICE	EO	6F0E	PUT	DD	8656
DIR	E1	6D86	RENUM	EA	7B92
DRAW	E9	7DF4	RSET	DC	85FC
DSKINI	D6	7753	SWAP	EB	7A03
DSKO\$	D7	88C5	UNLOAD	E4	822D
FIELD	DA	85BE	VERIFY	DF	6F29
FILES	E2	8172	WRITE	E3	8265
GET	DE	8657			

REMARQUE : On vérifiera que EXEC &H6D86 liste bien le catalogue.

D'autre part, le listage d'une instruction

```
10 PRINT:POKE &H8E58,&HD6 'ou D7,D8,etc...
```

donne bien DSKINI (ou DSKO\$, KILL, etc...) après que l'on ait fait RUN.

4. Les nouvelles fonctions du DOS sont les suivantes :

Nom	code hexa	Adresse hexa
CVD	FFAA	889B
CVI	FFA8	8895
CVS	FFA9	8898
DSKF	FFA7	73FA
DSKI\$	FFB2	8924
LOC	FFAE	742F

Nom	Code hexa	Adresse hexa
LOF	FFAF	7470
MKD\$	FFAD	88B5
MKI\$	FFAB	88AF
MKS\$	FFAC	88B2
SPACE\$	FFB0	79ED
STRING\$	FFB1	79CB

REMARQUE : De même que ci-dessus, l'instruction

```
10 PRINT X:POKE &H8E58,&HFF:POKE &H8E59,&HA7 'ou A8...
```

donne bien DSKF (ou CVI, etc...) après avoir fait RUN.

Création des nouvelles instructions Compilation

L'étude précédente va nous permettre d'adapter nos routines au TO9, ainsi qu'au TO7-70 équipé d'un lecteur de disquettes.

On obtiendra alors un BASIC comportant entre autres les procédures et les sprites décrits dans la troisième partie.

On pourra aussi effectuer la compilation de n'importe quel programme BASIC, qui sera rendu ainsi beaucoup plus rapide à l'exécution.

1. Fonctions de conversion radiants-degrés

La routine de la page 91 doit ici être modifiée puisque les fonctions utilisateurs FN sont autorisées avec le DOS.

On trouve en effet en \$623C une instruction JMP \$7A73 ; notre traitement sera donc en cas de rencontre du mot-clé FN de tester si l'on a FNR(ou FND(, pris alors en compte par notre routine ; sinon, on se branchera en \$7A73 pour le traitement normal.

La routine, placée ici en \$DC40, est la suivante :

DC40	DE	B9	LDU	\$B9	Adr.caract.FN->U
DC42	37	12	PULU	A,X	2 caract.après FN->X
DC44	CC	920	LDD	#\$920	Adr.traitmt. /->D
DC47	8C	4428	CMPX	#\$4428	A t'on "D(" ?
DC4A	27	0B	BEQ	\$DC57	Oui=>suite
DC4C	CC	25B6	LDD	#\$25B6	Adr.traitmt. *->D
DC4F	8C	5228	CMPX	#\$5228	A t'on "R(" ?
DC52	27	03	BEQ	\$DC57	Oui=>suite
DC54	7E	7A73	JMP	\$7A73	Non=>trait.normal FN
DC57	DF	B9	STU	\$B9	Sur 1er car.argument
DC59	34	06	PSHS	A,B	Adr.traitmt.->pile
DC5B	BD	07E0	JSR	\$07E0	Traite argument->FAC
DC5E...	a	... DC6E			=BA49 à BA59 pg.91
DC70	6E	F1	JMP	[,S++]	RAZ pile S;traitmt.

Nos deux nouvelles fonctions seront donc initialisées en plaçant l'adresse \$DC40 en \$623D.

II. Création de nouvelles instructions

1. Par rapport au BASIC 1.0 des TO7 et TO7-70, les instructions WHILE et WEND ont malheureusement été enlevées de la table des mots réservés ; il n'est donc plus possible de créer facilement une boucle WHILE-WEND (les routines \$16AC et 2F3, toujours présentes, imposant d'avoir &HAF et B0 comme codes de WHILE et WEND).

2. La routine correspondant à notre instruction SWAP ne se justifie plus puisque cette instruction existe dans le DOS (elle n'autorise toutefois pas le mélange des types) ; l'intérêt de INC est par contre toujours le même (deux fois plus rapide qu'une addition normale).

On remarquera que la valeur &H0001 est contenue en \$EA24 (\$F9CA pour le TO7) comme dans le cas du TO7-70.

3. Nos nouvelles instructions seront donc INC, CALL, PROC (et ARG, non exécutable) et SPRITE ; elles seront traitées grâce à une table des mots située en \$DB95 et à une table des adresses de traitement située en \$DB71 ; le nombre des nouveaux mots sera placé en \$DB70.

Il suffira pour cela d'*enlever du programme de création* des pages 93 et 106 les instructions 1320 à 1360, et de *modifier* ainsi les seules instructions 1010, 1310 et 1560 :

```
1010 CLEAR, &HDA00
1310 ADR0=&HDB71:ADR=ADR0+36 'Adresses tables
1560 ADR0=FIN+1:POKE &HDB70,NBRM
```

Bien entendu, les DATA des instructions 1700 à 2990 devront aussi être légèrement adaptées, comme on le verra ci-après.

4. Après exécution du programme de création ainsi modifié, il faudra bien sûr sauvegarder les routines et les tables sur disquette ; on fera donc :

```
SAVEM "ROUT", &HDA01, &HDFFF, &HDC01
```

L'activation des nouvelles instructions, qui devra être effectuée (en mode direct) avant toute utilisation des nouveaux mots, sera alors réalisée en faisant :

```
CLEAR, &HDA00:LOADM "ROUT", , R
```

III. Adaptation des routines

1. Initialisation

Les nouvelles instructions seront activées et traitées par la nouvelle routine d'initialisation suivante, compatible avec les informations obtenues au chapitre 2 :

DC0A	CC	DB95	LDD	#\$DB95	Adr.table nouv.mots
DC0D	FD	6216	STD	\$6216	
DC10	B6	DB70	LDA	\$DB70	Nombre nouv.mots->A
DC13	B7	6215	STA	\$6215	
DC16	8B	EC	ADDA	#\$EC	A+code 1er mot->A
DC18	C6	23	LDB	#\$23	Code de BLS
DC1A	FD	6BDB	STD	\$6BDB	Modif.\$6BDA...chap.2
DC1D	1F	50	TFR	PC,D	
DC1F	C3	0007	ADDD	#\$0007	#\$DC26->D
DC22	FD	6218	STD	\$6218	Adr.trait.nouv.x.mots
DC25	39		RTS		
DC26	8E	DB71	LDX	#\$DB71	Adr.table adresses
DC29	80	ED	SUBA	#\$ED	1er mot->0,etc...
DC2B	7E	2B38	JMP	\$2B38	Traitement mot

2. Les procédures

Les deux octets permettant la détection du déplacement d'un tableau sont ici situés en \$DFBE (au lieu de \$BFFE).

D'autre part, ARG est maintenant codé par EF, soit sur un octet au lieu de deux (code &HFFAB auparavant) ; ceci impose de modifier de la manière suivante la routine de la page 119 (les adresses données ici sont les anciennes):

BB50	9D	B8	JSR	\$B8	Code->A
BB52	88	84	BORA	#\$84	OU exclus.avec DIM
BB54	2B	09	BMI	\$BB5F	1->bit 7=>variable
BB56	26	20	BNE	\$BB78	A>0=>ARG
BB58	12		NOP		A=0=>DIM
BB59	3D	58	BSR	\$BBB3	SP+6(tableau)

En \$BB76, on écrira alors :

BB76	12		NOP		
BB77	12		NOP		
BB78	C6	EF	LDB	#\$EF	Code de ARG
BB7A	9D	D0	JSR	\$D0	A t'on ARG?

3. Les SPRITES

La zone permettant les échanges avec la mémoire écran sera ici placée en \$D400 ; la zone de gestion des sprites sera mise en \$DFCO.

L'octet permettant la détection d'une « collision » entre un sprite et une forme ou un autre sprite sera situé en \$DFBD au lieu de BFFD (on modifiera en conséquence les instructions 530 (et 400) et 280 (et 10) des pages 142 et 143.

Les routines \$F966 (EF15 pour le TO7-70) et F161 (\$F328) du TO7 (voir page 141) sont dans le cas du TO9 situées respectivement en \$F85D et EF1D à signaler qu'en cas de modification ultérieure du moniteur, ces routines seront immédiatement retrouvées en étudiant les routines d'affichage PLOT\$ et PUTCS.

Enfin, le TO9 et le TO7-70 permettent l'affichage de 16 couleurs au lieu des seules 8 tolérées par le BASIC 1.0 ; ceci impose de toujours mettre à 1 les bits 6 et 7 de la mémoire de couleur, ce qui sera fait en ajoutant une instruction *ORA # \$C0* entre \$BD92 et BD93 ; on devra donc aussi *décaler de 2* les deux branchements au sous-programme (en \$BCFC et BD34).

4. Résumé

1. Tous les DATA à écrire sont listés ci-après, avec les modifications à effectuer par rapport aux pages 146 et 147 :

```
2000 DATA Initialisations, &HDC01, CC, DC40, FD, 623D.
2010 DATA BD, DE60
2020 DATA CC, DB95, FD, 6216, B6, DB70, B7, 6215, 8B, EC, C6, 23, F
D, 6BDB, 1F, 50, C3, 007, FD, 6218, 39
2030 DATA 8E, DB71, 80, ED, 7E, 2B38, FIN
2100 DATA Fonctions, &HDC40, DE, B9, 37, 12, CC, 920, 8C, 4428, 2
7, B, CC, 25B6, 8C, 5228, 27, 3, 7E, 7A73, DF, B9, 34, 6, BD, 7E0, BD, 2
52B, BD, 1C81, CC, 7B8E, DD, 55, F, 5D, CC, FA35, DD, 57, 6E, F1, FIN
2200 DATA INC, &HDC90, BD, A48... 3, 8E, EA24, BD, 1C08, etc...
2400 DATA CALL, &HDC00, C6, 7, BD, 336, 9E, 22, BF, DFBE, 109E, B9
... 9D, B8, 88, 84, 2E, 9, 26, 20, 12, 8D, 58, 8A, C0, etc...
2405 DATA 12, 12, C6, EF, 9D, D0, ... 22, BC, DFBE, 26, 34, etc...
2410 DATA 9D, B8, 81, 84... 27, 3, 7A, DFBE, 96, 5, F, 7, 5D, 39, FIN
2500 DATA PROC, &HDD7C, 81, 80, 27, 6C, 5D, 7E6, etc...
2510, 2520 et 2530 DATA inchangés
2600 DATA SPRITE, &HDE60, 8E, DFC2, 86, 1F, A7, 81, 4A, 26, FB, 39
```

2610 DATA 86,FF,B7,DFBD,BD,815...58,58,C3,DFC0,DD,63,5C
 ...C6,60,3D,C3,D400,DD,67
 2620 DATA EC,9F6165...27,8,17,0C1,6C,9F6165,39
 2630 DATA FC,6576...10BE,6578,BD,F85D,D,5C,26,7,34,50,1
 7,089,35,50,9F,58,109E,67,BD,EF1D,86,10,97,55
 2640 DATA D6,56...27,3,7F,DFBD,5A,26,E4,30,8825,etc...
 2650 DATA 86,10,97,55...48,48,48,8A,C0,97,56,C6,3,etc..
 2660 DATA 9E,5A,109E,5C,BD,F85D,9F,58,109E,67,BD,EF1D,8
 6,10,97,55,etc...
 2950 DATA FIN des routines
 2990 DATA CALL,&HDC00,PROC,&HDD7C,ARG,&H7F3,SPRITE,&HDE
6B,INC,&HDC90,FIN

On constate que l'on devra aussi supprimer les anciennes instructions 1710, 2095 à 2110 et 2295 à 2300 (et 1320 à 1360 : voir II) ; de plus, on a renuméroté ici les instructions 1700 à 2000 pour raison de compatibilité avec les routines de compilation.

2. Le résultat du programme de création est le suivant :

ROUTINE Initialisations		
Début:DC01	Fin:DC2D	Somme: 6163
ROUTINE Fonctions		
Début:DC40	Fin:DC71	Somme: 5663
ROUTINE INC		
Début:DC90	Fin:DCD1	Somme: 5863
ROUTINE CALL		
Début:DCE0	Fin:DD7B	Somme: 15347
ROUTINE PROC		
Début:DD7C	Fin:DE56	Somme: 22239
ROUTINE SPRITE		
Début:DE60	Fin:DFB5	Somme: 38921

IV. Compilation

1. La première modification à effectuer consiste à déplacer les routines (implantées ici en \$DA01 au lieu de BD01) et à modifier l'adresse où est rangée la valeur du déplacement des adresses de variables (\$DB6E au lieu de BFFB).

2. Il faut aussi modifier le déplacement des adresses de branchement ; les adresses effectives de ceux-ci sont en effet maintenant comprises entre \$8E53 et DFFF, au lieu de 65F4 à DFFF ; le déplacement sera donc ici de &H1AE = &H9001-8E53 (au lieu de &H2A0D).

Il faudra donc tester en \$BE1D si le premier octet de l'adresse effective est égal à &HC3 (on aurait alors C4, interdit) au lieu de 9A.

REMARQUE :

On pourrait faire calculer automatiquement le déplacement des adresses de branchement lors de l'initialisation, comme on le fait pour les adresses de variables.

On calculerait pour cela \neq \$9002-[\$611C].

3. Par rapport aux DATA de la page 181, nous avons ici choisi de renuméroter à partir de 1700 pour pouvoir créer les routines correspondantes en même temps que les nouvelles instructions ; les modifications sont les suivantes :

1700 DATA Initialisation, &HDA01, CC, DA28, FD, 6271, CC, DAA1, FD, 627D, CC, DB07, FD, 6298... FD, DE6E, 39, FIN Init.
1800 DATA Branchements, &HDA28, 27, 8, 81, 87, etc...
1810 DATA 32, 62, 9D, B2, etc...
1820 DATA 9D, B2, 31, 41... 1F, 10, 81, C3, 27, 5, CC, 1AE, 3D, 46, 3F, B9, 39, EC, A4, 83, 1AE, DD, B9, 39, FIN Branchements
1850 DATA Trait. opérandes, &HDAA1, 32, 62, 9E, B9, etc...
1860 DATA 31, 22, 109C, B9, etc...
1900 DATA Rech. variables, &HDB07, 109E, B9... DD, 3F, FC, DE6E, C, 3C, D0, 3C, etc...
1910 DATA AE, A1, 109F, B9, FC, DE6E, 43... 9F, 3D, F, 7, 39, FIN

4. Le résultat du programme de création est alors le suivant :

ROUTINE Initialisation		
Début: DA01	Fin: DA26	Somme: 5520
ROUTINE Branchements		
Début: DA28	Fin: DAA0	Somme: 12633
ROUTINE Trait.opérandes		
Début: DAA1	Fin: DB06	Somme: 8628
ROUTINE Rech.variables		
Début: DB07	Fin: DB6D	Somme: 10309

Annexe 1

Routines du moniteur

<i>Adresse</i>	<i>Nom</i>	<i>Action</i>
\$E800	INIT\$	Initialisation de l'affichage
\$E803	PUTC\$	Affichage du caractère contenu dans B ; gestion des attributs d'écran
\$E806	GETC\$	Lecture du clavier (code touche retourné dans B)
\$E809	KTST\$	Lecture rapide du clavier (bit C du CC mis à 1 si une touche est enfoncée)
\$E80C	DRAW\$	Tracé du segment de droite d'extrémité (X,Y)
\$E80F	PLOT\$	Affichage du point de coordonnées (X,Y)
\$E812	RSCOS\$	Gestion de l'interface de communication
\$E815	K7COS\$	Entrée/sortie sur cassette
\$E818	GETL\$	Lecture du crayon optique (retourne coordon- nées dans X et Y, avec bit C du CC mis à 0)

<i>Adresse</i>	<i>Nom</i>	<i>Action</i>
\$E81B	LPIN\$	Lecture bouton du crayon optique (bit C du CC mis à 1 si enfoncé)
\$E81E	NOTES	Génération de la note de musique contenue dans B
\$E821	GETP\$	Lecture de la couleur du point (X,Y) (retournée dans B)
\$E824	GETS\$	Lecture du caractère situé en (X,Y) (1 à 40,0 à 24 ; code ASCII retourné dans B)
\$E827	JOYSS	Lecture de la manette de jeu dont numéro dans A (direction retournée dans B ; bit C du CC mis à 1 si bouton enfoncé)
\$E82A	DKCO\$	Entrée/sortie sur disquette
\$E82D	MENU\$	Retour au menu principal (par JMP)
\$E830	KBIN\$	Sortie (par JMP) d'un programme d'interruption
\$E833	CHPL\$	Écriture du point " caractère " de coordonnées X et Y (1 à 40, 0 à 24)

REMARQUE : A,B,CC, X et Y désignent les registres du 6809.

Annexe 2

Principales adresses du moniteur

1. Page 0

<i>Adresse</i>	<i>Nom</i>	<i>Rôle</i>
\$6019	STATUS	bit 7 : semi-graphique ; bit 6 : scroll rapide ; bit 5 : interruptions utilisateur ; bit 4 : graphiques sans couleur (T07-70) ; bit 3 : lecture clavier ; bit 2 : curseur visible ou invisible ; bit 0 : touche clavier déjà lue
\$6027	TIMEPT	Pointeur (2 octets) sur traitement des interruptions timer utilisateur
\$602D	USERAF	Pointeur (2 octets) sur générateur de caractères utilisateurs
\$602F	SWI1	Pointeur (2 octets) sur traitement des interruptions logicielles (SWI)
\$6038	FORME	Code de la couleur pour affichages graphiques (T07:-8 à +7 ; T07-70:-8 à +15)

<i>Adresse</i>	<i>Nom</i>	<i>Rôle</i>
\$603B	COLOUR	Couleurs courantes ; bits 0,1,2 : couleur fond ; bits 3,4,5 : couleur forme ; bits 6,7 : couleurs pastels (TO7-70)
\$603D	PLOTX	Abscisse (2 octets) du dernier point affiché
\$603F	PLOTY	Ordonnée (2 octets) du dernier point affiché
\$6041	CHDRAW	0, ou code ASCII du caractère à afficher (par PLOT\$, DRAW\$ ou CHPL\$)
\$6060	STADR	Adresse (2 octets) du 1 ^{er} octet de la fenêtre écran
\$6062	ENDDR	Adresse + 1 (2 octets) du dernier octet de la fenêtre
\$6073	BUZZ	Sémaphore du bip clavier pour TO7-70
\$60CD	PTCLAV	Pointeur (2 octets) sur la table de décodage du clavier pour TO7-70
\$60CF	PTGENE	Pointeur (2 octets) sur le générateur de caractères standards pour TO7-70

2. Adresses d'entrées/sorties

PIA système 6846 :\$E7C0 à \$E7C7

\$E7C3 (PRC) : registre de données

bit 0 : commutation mémoire écran "forme" (1) et "couleur" (0)

bit 1 : interrupteur crayon optique

bit 2 : couleur pastel du tour pour TO7-70

bit 3 : affichage en minuscules

bit 4,5,6 : couleur du tour

bit 7 : lecture cassette

\$E7C6 (TMSB-TLSB) : valeur du timer (2 octets)

PIA système 6821 : \$E7C8 à \$E7CB

\$E7C8 (PRA) : registre de données du port A ; lecture matrice clavier

\$E7C9 (PRB) : registre de données du port B ; écriture matrice clavier
bits 0,1,2 : multiplexage clavier pour TO7-70
bits 3 à 7 : sélection banques mémoire pour TO7-70

PIA jeux 6821 : \$E7CC à \$E7CF (manettes de jeu)

Interface de communication 6821 : \$E7E0 à \$E7E3

Gate-Array (TO7-70) : \$E7E4 à \$E7E7

Annexe 3

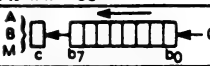
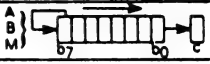
Les instructions du 6809

1. Instructions de branchement

Instruction	Forme	Adressing Mode Relative			Description	5 H	3 N	2 Z	1 V	0 C
		OIP	—	#						
BCC	BCC LBCC	24	3	2	Branch C=0	•	•	•	•	•
		10	5(6)	4	Long Branch C=0	•	•	•	•	•
		24								
BCS	BCS LBCS	25	3	2	Branch C=1	•	•	•	•	•
		10	5(6)	4	Long Branch C=1	•	•	•	•	•
		25								
BEQ	BEQ LBEQ	27	3	2	Branch Z=0	•	•	•	•	•
		10	5(6)	4	Long Branch Z=0	•	•	•	•	•
		27								
BGE	BGE LBGE	2C	3	2	Branch ≥ Zero	•	•	•	•	•
		10	5(6)	4	Long Branch ≥ Zero	•	•	•	•	•
		2C								
BGT	BGT LBGT	2E	3	2	Branch > Zero	•	•	•	•	•
		10	5(6)	4	Long Branch > Zero	•	•	•	•	•
		2E								
BHI	BHI LBHI	22	3	2	Branch Higher	•	•	•	•	•
		10	5(6)	4	Long Branch Higher	•	•	•	•	•
		22								
BHS	BHS LBHS	24	3	2	Branch Higher or Same	•	•	•	•	•
		10	5(6)	4	Long Branch Higher or Same	•	•	•	•	•
		24								
BLE	BLE LBLE	2F	3	2	Branch ≤ Zero	•	•	•	•	•
		10	5(6)	4	Long Branch ≤ Zero	•	•	•	•	•
		2F								
BLO	BLO LBLO	25	3	2	Branch lower	•	•	•	•	•
		10	5(6)	4	Long Branch Lower	•	•	•	•	•
		25								

Instruction	Forme	Adressing Mode			Description	5	3	2	1	0
		Relative				H	N	Z	V	C
		OP	—	#						
BLS	BLS	23	3	2	Branch Lower or Same	•	•	•	•	•
	LBLS	10 23	5(6)	4	Long Branch Lower or Same	•	•	•	•	•
BLT	BLT	2D	3	2	Branch < Zero	•	•	•	•	•
	LBLT	10 2D	5(6)	4	Long Branch < Zero	•	•	•	•	•
BMI	BMI	2B	3	2	Branch Minus	•	•	•	•	•
	LBMI	10 2B	5(6)	4	Long Branch Minus	•	•	•	•	•
BNE	BNE	2B	3	2	Branch Z ≠ 0	•	•	•	•	•
	LBNE	10 2B	5(6)	4	Long Branch Z ≠ 0	•	•	•	•	•
BPL	BPL	2A	3	2	Branch Plus	•	•	•	•	•
	LBPL	10 2A	5(6)	4	Long Branch Plus	•	•	•	•	•
BRA	BRA	20	3	2	Branch Always	•	•	•	•	•
	LBRA	16	5	3	Long Branch Always	•	•	•	•	•
BRN	BRN	21	3	2	Branch Never	•	•	•	•	•
	LBRN	10 21	5	4	Long Branch Never	•	•	•	•	•
BSR	BSR	8D	7	2	Branch to Subroutine	•	•	•	•	•
	LBSR	17	9	3	Long Branch to Subroutine	•	•	•	•	•
BVC	BVC	2B	3	2	Branch V = 0	•	•	•	•	•
	LBVC	10 2B	5(6)	4	Long Branch V = 0	•	•	•	•	•
BVS	BVS	29	3	2	Branch V = 1	•	•	•	•	•
	LBVS	10 29	5(6)	4	Long Branch V = 1	•	•	•	•	•

2. Instructions

Instruction	Forme	Adressing Modes												Description	5	3	2	1	0								
		Immediate			Direct			Indexed			Extended									Inherent							
		Op	-	#	Op	-	#	Op	-	#	Op	-	#							Op	-	#					
ABX														3A	3	1	B + X - X (Unsigned)					*	*	*	*	*	
ADC	ADCA ADCB	88 C9	2	2	2	99 D9	4	2	2	A9 E9	4+	2+	2+	B9 F9	5	3		A + M + C - A B + M + C - B					1	1	1	1	1
ADD	ADDA ADDB ADDD	88 C8 C3	2	2	2	98 D8 D3	4	2	2	A8 E8 E3	4+	2+	2+	B8 F8 F3	5	3		A + M - A B + M - B D + M: M + 1 - D					1	1	1	1	1
AND	ANDA ANDB ANDCC	84 C4 1C	2	2	2	94 D4	4	2	2	A4 E4	4+	2+	2+	B4 F4	5	3		A & M - A B & M - B CC & IMM - CC					*	1	1	0	*
ASL	ASLA ASLB ASL														48 58	2	1						8	1	1	1	1
ASR	ASRA ASRB ASR														47 57	2	1						8	1	1	*	1
BIT	BITA BITB	85 C5	2	2	2	95 D5	4	2	2	A5 E5	4+	2+	2+	B5 F5	5	3		Bit Test A (M A A) Bit Test B (M A B)					*	1	1	0	*
CLR	CLRA CLRB CLR														4F 5F	2	1	0 - A 0 - B 0 - M					*	0	1	0	0
CMP	CMPA CMPB CMPD CMPM CMPS CMPU CMPX CMPY	81 C1 10 83 11 8C 11 83 8C 10	2	2	2	91 D1 10 93 11 9C 11 93 8C 10	4	2	2	A1 E1 10 A3 11 AC 11 A3 8C 10	4+	2+	2+	B1 F1 10 B3 11 BC 11 BC 10	5	3		Compare M from A Compare M from B Compare M M + 1 from D Compare M M + 1 from S Compare M M + 1 from U Compare M M + 1 from X Compare M M + 1 from Y					8	1	1	1	1
COM	COMA COMB COM														43 53	2	1	A - A B - B M - M					*	1	1	0	1
CWA		3C	2	2	2													CC A IMM - CC Wait for interrupt									7
DAA															19	2	1	Decimal Adjust A					*	1	1	0	1
DEC	DECA DECB DEC														4A 5A	2	1	A - 1 - A B - 1 - B M - 1 - M					*	1	1	1	*
EOR	EGRA EORB	88 C8	2	2	2	98 D8	4	2	2	A8 E8	4+	2+	2+	B8 F8	5	3		A ^ M - A B ^ M - B					*	1	1	0	*
EXG	R1, R2	1E	8	2														R1 - R2					*	*	*	*	*
INC	INCA INCB INC														4C 5C	2	1	A + 1 - A B + 1 - B M + 1 - M					*	1	1	1	*
JMP																		EA ³ - PC					*	*	*	*	*
JSR																		Jump to Subroutine					*	*	*	*	*
LCA	LDA LDB LDD LDS LDU LDX LDY	86 C6 CC 10 CE 8E 10 8E	2	2	2	96 D6 DC 10 DE 9E 10 9E	4	2	2	A6 E6 5 6 EE 5 AE 6 AE	4+	2+	2+	B6 F6 5 6 FE 5 BE 6 BF	5	3		M - A M - B M M + 1 - D M M + 1 - S M M + 1 - U M M + 1 - X M M + 1 - Y					*	1	1	0	*
LEA	LEAS LEAL LEAX LEAY														32 33 30 31	4+	2+	EA ³ - S EA ³ - U EA ³ - X EA ³ - Y					*	*	*	*	*

Op: code opération en hexadécimal

#: nombre de cycles

#: nombre d'octets

●: flag non affecté (H,N,Z,V,C)

†, Ø ou 1: flag affecté

+ (adressage indexé) = Ø à 8 cycles, et Ø à 2 octets à ajouter selon le post-octet (voir annexe 4)

Instruction	Forme	Adressing Modes												Description	5	3	2	1	0			
		Immediate			Direct			Indexed			Extended									Inherent		
		Op	-	#	Op	-	#	Op	-	#	Op	-	#							Op	-	#
LSL	LSLA LSLB LSL				08	6	2	68	6+	2+	78	7	3	48 58	2 2	1 1						
LSR	LSRA LSRB LSR				04	6	2	64	6+	2+	74	7	3	44 54	2 2	1 1						
MUL														3D	11	1	A × B - D (Unsigned)				9	
NEG	NEGA NEGB NEG				00	6	2	60	6+	2+	70	7	3	40 50	2 2	1 1	A + 1 - A B + 1 - B M + 1 - M	8 8 8	1 1 1	1 1 1	1 1 1	
NOP														12	2	1	No Operation					
OR	ORA ORB ORCC	8A CA 1A	2 2 3	2 2 2	9A DA EA	4 4 +	2 2 +	AA +	2+ 2+	BA FA	5 5	3 3					A V M - A B V M - B CC V IMM - CC		1 1 1	0 0 7	0 0 7	
PSH	PSHS PSHU	34 36	5+ 5+	4 4	2 2												Push Registers on S Stack Push Registers on U Stack					
PUL	PULS PULU	35 37	5+ 5+	4 4	2 2												Pull Registers from S Stack Pull Registers from U Stack					
ROL	ROLA ROLB ROL				09	6	2	69	6+	2+	79	7	3	49 59	2 2	1 1						
ROR	RORA RORB ROR				06	6	2	66	6+	2+	76	7	3	46 56	2 2	1 1						
RTI														38	6/15	1	Return From Interrupt				7	
RTS														39	5	1	Return from Subroutine					
SBC	SBCA SBCB	82 C2	2 2	2 2	92 D2	4 4	2 2	A2 E2	4+ 4+	2+ 2+	B2 F2	5 5	3 3				A - M - C - A B - M - C - B	8 8	1 1	1 1	1 1	
SEX														1D	2	1	Sign Extend B into A		1	1	0	
ST	STA STB STD STS				97 D7 DD	4 4 5	2 2 2	A7 E7 ED	4+ 4+ 5+	2+ 2+ 2+	B7 F7 FD	5 5 6	3 3 3				A - M B - M D - M M + 1 S - M M + 1		1 1 1 1	0 0 0 0		
	STU STX STY				DF DF 9F 10 9F	5 5 5 6	2 2 2 3	EF EF AF 10 AF	5+ 5+ 5+ 6+	2+ 2+ 2+ 3+	FF FF BF 10 BF	6 6 6 7 6	3 3 3 4 3				U - M M + 1 X - M M + 1 Y - M M + 1		1 1 1	0 0 0		
SUB	SUBA SUBB SUBD	80 C0 83	2 2 4	2 2 3	90 D0 93	4 4 6	2 2 2	A0 E0 A3	4+ 4+ 6+	2+ 2+ 2+	B0 F0 B3	5 5 7	3 3 3				A - M - A B - M - B D - M M + 1 - D	8 8 8	1 1 1	1 1 1	1 1 1	
SWI	SWI ⁶ SWI ²⁶ SWI ³⁶													3F 10 3F 11 3F	19 20 20 20 20	1 2 1 1 1	Software Interrupt 1 Software Interrupt 2 Software Interrupt 3					
SYNC														13	≥4	1	Synchronize to Interrupt					
TFR	R1, R2	1F	6	2													R1 - R2 ²					
TST	TSTA TSTB TST				0D	6	2	6D	6+	2+	7D	7	3	4D 5D	2 2	1 1	Test A Test B Test M		1 1 1	0 0 0		

Notes:

- 3: EA est l'adresse effective ("effective adress")
- 4: 5 cycles pour PSH et PUL, plus 1 cycle pour chaque octet transféré
- 5(6): Branchements longs; 6 cycles si branchement effectué, 5 sinon
- 6: SWI positionne I et F à 1 (pas SWI2 ou SWI3)
- 8: Valeur de H non définie
- 9: cas spécial pour MUL: 1→C si bit 7=1

Annexe 4

Adressages indexés et relatifs – conversions décimal/hexa

1. Adressages indexés

212

Type	Forms	Non Indirect			Indirect		
		Assembler Form	Postbyte Op Code	+ - #	Assembler Form	Postbyte Op Code	+ - #
Constant Offset From R (2's Complement Offsets)	No Offset	,R	1RR00100	0 0	[,R]	1RA10100	3 0
	5 Bit Offset	n, R	0RAnnnnn	1 0	defaults to 8-bit		
	8 Bit Offset	n, R	1RR01000	1 1	[n, R]	1RA11000	4 1
	16 Bit Offset	n, R	1RR01001	4 2	[n, R]	1RA11001	7 2
Accumulator Offset From R (2's Complement Offsets)	A Register Offset	A, R	1RR00110	1 0	[A, R]	1RA10110	4 0
	B Register Offset	B, R	1RR00101	1 0	[B, R]	1RA10101	4 0
	D Register Offset	D, R	1RR01011	4 0	[D, R]	1RA11011	7 0
	Increment By 1	,R +	1RR00000	2 0	not allowed		
Auto Increment/Decrement R	Increment By 2	,R + +	1RR00001	3 0	[,R + +]	1RA10001	6 0
	Decrement By 1	, - R	1RR00010	2 0	not allowed		
	Decrement By 2	, - - R	1RR00011	3 0	[, - - R]	1RA10011	6 0
	8 Bit Offset	n, PCR	1xx01100	1 1	[n, PCR]	1xx11100	4 1
Constant Offset From PC (2's Complement Offsets)	16 Bit Offset	n, PCR	1xx01101	5 2	[n, PCR]	1xx11101	8 2
	16 Bit Address	-	-	-	[n]	10011111	5 2
Extended Indirect							

R : X,Y,U ou S

RR : 00 ↔ X

01 ↔ Y

10 ↔ U

11 ↔ S

x : indifférent

~ : NOMBRE DE CYCLES SUPPLÉMENTAIRES

: NOMBRE D'OCTETS SUPPLÉMENTAIRES

3. Conversion décimal/hexa

<i>Chiffre hexadécimal</i>							
4		3		2		1	
HEXA	DEC.	HEXA	DEC.	HEXA	DEC.	HEXA	DEC.
0	0	0	0	0	0	0	0
1	16	1	256	1	16	1	1
2	32	2	512	2	32	2	2
3	48	3	768	3	48	3	3
4	64	4	1024	4	64	4	4
5	80	5	1280	5	80	5	5
6	96	6	1536	6	96	6	6
7	112	7	1792	7	112	7	7
8	128	8	2048	8	128	8	8
9	144	9	2304	9	144	9	9
A	160	A	2560	A	160	A	10
B	176	B	2816	B	176	B	11
C	192	C	3072	C	192	C	12
D	208	D	3328	D	208	D	13
E	224	E	3584	E	224	E	14
F	240	F	3840	F	240	F	15

Conversion décimal/hexa — Branchements relatifs longs

Annexe 5

Code ASCII

<i>Code décimal</i>	<i>Code hexa</i>		<i>Code décimal</i>	<i>Code hexa</i>		<i>Code décimal</i>	<i>Code hexa</i>	
0	00	NUL	42	2A	*	85	55	U
1	01		43	2B	+	86	56	V
2	02	STOP clavier	44	2C	,	87	57	W
3	03	CONT/C clavier	45	2D	—	88	58	X
4	04		46	2E	.	89	59	Y
5	05		47	2F	/	90	5A	Z
6	06		48	30	0	91	5B	[
7	07	SONNETTE	49	31	1	92	5C	\
8	08	BS(←clavier)	50	32	2	93	5D]
9	09	HT(→clavier)	51	33	3	94	5E	↑
10	0A	LF(↓ clavier)	52	34	4	95	5F	—
11	0B	VT(↑ clavier)	53	35	5	96	60	—
12	0C	RAZ clavier	54	36	6	97	61	a
13	0D	ENTREE clavier	55	37	7	98	62	b
14	0E	SO (semi graphique)	56	38	8	99	63	c
15	0F	SI (alphanumérique)	57	39	9	100	64	d

<i>Code décimal</i>	<i>Code hexa</i>		<i>Code décimal</i>	<i>Code hexa</i>		<i>Code décimal</i>	<i>Code hexa</i>	
16	10		58	3A	:	101	65	e
17	11	DC1 (clignot. curseur)	59	3B	;	102	66	f
18	12	DC2 (répétition)	60	3C	<	103	67	g
19	13		61	3D	=	104	68	h
20	14	DC4 (arrêt curseur)	62	3E	>	105	69	i
21	15		63	3F	?	106	6A	j
22	16	SS2 (car. accentués)	64	40	@	107	6B	k
23	17		65	41	A	108	6C	l
24	18	CONT/X clavier (efface fin ligne)	66	42	B	109	6D	m
25	19		67	43	C	110	6E	n
26	1A		68	44	D	111	6F	o
27	1B	ESC	69	45	E	112	70	p
28	1C	INS clavier	70	46	F	113	71	q
29	1D	EFF clavier	71	47	G	114	72	r
30	1E	↵ clavier	72	48	H	115	73	s
31	1F	US	73	49	I	116	74	t
32	20	Espace	74	4A	J	117	75	u
33	21	!	75	4B	K	118	76	v
34	22	"	76	4C	L	119	77	w
35	23	#	77	4D	M	120	78	x
36	24	\$	78	4E	N	121	79	y
37	25	%	79	4F	O	122	7A	z
38	26	&	80	50	P	123	7B	{
39	27	'	81	51	Q	124	7C	
40	28	(82	52	R	125	7D	}
41	29)	83	53	S	126	7E	—
			84	54	T	127	7F	■

Annexe 6

Instructions BASIC

<i>Nom</i>	<i>Code hexa</i>	<i>Adresse hexa</i>
ATTRB	A8	3316
AUTO	9A	131D
BEEP	A3	35E6
BOX	A6	3507
BOXF	A646	3507
CLEAR	AE	57B
CLOSE	B6	2DD9
CLS	9D	35EB
COLOR	A4	33D5
CONSOLE	9E	3353
CONT	AC	56B
DATA	83	663
DEF	A9	1801
DEFDBL	95	12B5
DEFINT	93	12AF

<i>Nom</i>	<i>Code hexa</i>	<i>Adresse hexa</i>
MERGE	B4	2E91
MID\$	FF9C	1100
MOTOR	A0	3AF8
NEW	B1	4B6
NEXT	82	1605
OFF	C3	3AF8
ON	96	36B5
OPEN	B5	2E0D
PEN	B8	3635
PLAY	B9	3F14
POKE	AA	F1F
PRINT	AB	F78
PSET	9F	34EC
READ	85	2805
REM	8C	666

<i>Nom</i>	<i>Code hexa</i>	<i>Adresse hexa</i>
DEFSNG	94	12B2
DEFSTR	92	12AC
DELETE	9B	12F0
DIM	84	A01
ELSE	(3A)8F	6BE
END	80	53B
ERL	C1	7C2
ERR	C2	7B6
ERROR	98	1799
EXEC	A2	378F
FOR	81	1578
GO	87	606
IF	89	697
INPEN	B7	36E7
INPUT	FFA1	27A5
LET	86	722
LINE	A5	34F5
LIST	AD	2C74
LOAD	B3	2E99
LOCATE	9C	32E2

<i>Nom</i>	<i>Code hexa</i>	<i>Adresse hexa</i>
	(3A)8D	666
RESTORE	8A	527
RESUME	99	17A4
RETURN	8B	640
RUN	88	5F2
SAVE	B2	2CB3
SCREEN	FFA4	33CC
SKIPF	A1	39C1
STEP	C6	15D7
STOP	8E	544
SUB	BC	60E
THEN	C4	6AC
TO	BB	60A
TROFF	91	139F
TRON	90	139E
UNMASK	A7	3453
WAIT	97	F29
WEND	B0	6236
WHILE	AF	6233

REMARQUE: La table des noms d'instructions commence en \$0092, celle des adresses en \$26B (utilisée en \$2B25).

Les instructions MID\$, INPUT et SCREEN sont décodées en \$2B4E.

Les variables ERL et ERR sont traitées par la routine \$770.

Annexe 7

Fonctions BASIC

<i>Nom</i>	<i>Code hexa</i>	<i>Adresse hexa</i>
ABS	FF82	1CD5
ASC	FF8E	E4F
CDBL	FF93	2518
CHR\$	FF8F	E3B
CINT	FF91	24C3
COS	FF87	2689
CSNG	FF92	252B
CSRLIN.	FFA2	35F7
EOF	FF90	3066
EXP	FF86	23FB
FIX	FF94	1D61
FN	BD	623C
FRE	FF83	C36
GR\$	FF99	3781
HEX\$	FF95	1164

<i>Nom</i>	<i>Code hexa</i>	<i>Adresse hexa</i>
MID\$	FF9C	E7E
OCT\$	FF96	1165
PEEK	FF8A	F15
POINT	FFA3	345A
POS	FFA5	30D4
PTRIG	FFA6	3630
RIGHT\$	FF9B	E77
RND	FF9F	2470
SCREEN	FFA4	3462
SGN	FF80	1CBE
SIN	FF88	268F
SPC(BE	101F
SQR	FF84	2388
STICK	FF97	3617
STRIG	FF98	3627

<i>Nom</i>	<i>Code hexa</i>	<i>Adresse hexa</i>
INKEY\$	FFA0	35FD
INPUT	FFA1	3099
INSTR	FF9D	1058
INT	FF81	1D71
LEFT\$	FF9A	E5A
LEN	FF8B	E32
LOG	FF85	19E0

<i>Nom</i>	<i>Code hexa</i>	<i>Adresse hexa</i>
STR\$	FF8C	C64
TAB(BA	F41
TAN	FF89	26DA
USING	BF	1403
USR	C0	182C
VAL	FF8D	EC3
VARPTR	FF9E	17EA

REMARQUE: La table des noms des fonctions commence en \$1CF, celle des adresses en \$0020 (utilisée en \$2A93).

FN et USR sont décodés par la routine \$770.

SPC,TAB et USING sont traitées par l'instruction PRINT (\$F78).

Annexe 8

Les Opérateurs BASIC

<i>Opérateur</i>	<i>Symbole</i>	<i>Code hexa</i>	<i>Priorité hexa</i>	<i>Adresse hexa</i>	<i>Notes</i>
Concaténation	+	C7		DBB	Décodée en \$85B ; pour chaînes
Puissance	↑	CB	7F	2391	Opérandes traités en \$8F3
— unaire	—	C8	7D	25F3	Décodé par \$770 (en \$789)
Division	/	CA	7C	920	
Multiplication	*	C9	7C	25B6	
Division entière	@	D2	7B	2635	
Reste	MOD	D1	7A	265C	
Addition	+	C7	79	2590	
Soustraction	—	C8	79	2582	
Supérieur à	>	D3	64	266D	Traité en \$97A (par \$99C)
Egal à	=	D4	64	266D	Idem

<i>Opérateur</i>	<i>Symbole</i>	<i>Code hexa</i>	<i>Priorité hexa</i>	<i>Adresse hexa</i>	<i>Notes</i>
Sup.ou égal à	> =	D3D4	64	266D	Idem
Inférieur à	<	D5	64	266D	Idem
Différent de	<>	D5D3	64	266D	Idem
Inf. ou égal à	< =	D5D4	64	266D	Idem
Complément	NOT	C5	5A	7A3	Décodé par \$770 (en \$79F)
ET	AND	CC	50	9E5	
OU	OR	CD	46	9EA	
OU exclusif	XOR	CE	3C	9EF	
Equivalence	EQV	CF	32	9F4	
Implication	IMP	D0	28	9F9	

REMARQUE: La table des opérateurs (priorité et adresse) commence en \$006E.

Le — unaire et NOT sont traités par la routine \$770.

Les opérateurs de relation sont décodés en \$834 (routine \$81A).

Annexe 9

Principales adresses du BASIC

<i>Adresse</i>	<i>Nombre d'octets</i>	<i>Commentaire sur le contenu</i>
\$6102	1	Nombre d'indices d'un tableau
\$6104	1	différent de 0 pour l'instruction DIM
\$6105	1	type d'une valeur
\$6107	1	contient 1 si on veut l'adresse du début d'un tableau
\$611C	2	Adresse 1 ^{ère} instruction du programme (\$65F5)
\$611E	2	Adresse début zone des variables
\$6120	2	Adresse début zone des tableaux
\$6122	2	Adresse début zone libre
\$6124	2	Adresse du " fond " de la pile système
\$612A	2	Adresse fin zone des chaînes
\$612C	2	Numéro de la ligne courante du programme (ou &HFFFF si on est en mode direct)
\$6130	2	Valeur d'une étiquette de branchement
\$6138	2	Pointeur de ligne de DATA
\$613D	2	Adresse de la valeur d'une variable
\$613F	2	Adresse où l'on doit affecter une valeur

<i>Adresse</i>	<i>Nombre d'octets</i>	<i>Commentaire sur le contenu</i>
\$6155	8	Accumulateur flottant (FAC)
\$615D	1	Signe de la valeur située dans FAC
\$6163	9	Second accumulateur flottant (et signe)
\$6186	1	Différent de 0 en mode trace (TRON)
\$618C	2	Adresse du sommet de la pile (valeur de S)
\$61A2	1	Différent de 0 en mode protégé
\$61B9	2	Adresse du caractère courant du programme
\$61CA	3	Contient JMP \$7E9 (vérifie qu'on a une virgule)
\$61CD	3	Contient JMP \$2502 (Détermine le type d'une valeur)
\$61D0	3	Contient JMP \$7EB (vérifie la syntaxe)
\$6201	1	Nombre des instructions et opérateurs BASIC
\$6202	2	Adresse de la table des instructions
\$6204	2	Adresse table des adresses de traitement des instructions
\$6206	1	Nombre des fonctions BASIC
\$6207	2	Adresse de la table des fonctions
\$6209	2	Adresse de la table des adresses de traitement des fonctions
\$6233	3	Traitement de WHILE (JMP \$7F3 : SN Error)
\$6236	3	Traitement de WEND (Idem)
\$6239	3	Traitement de DEFFN (Idem)
\$623C	3	Traitement de FN (Idem)
\$626D	3*20	Points de contrôles de diverses routines
\$62AC	1	Nombre de caractères dans le buffer d'E/S (cassette)
\$62AD	2	Adresse du caractère courant du buffer d'E/S
\$62B2	255	Buffer d'E/S (cassette)
\$6445	255	Buffer d'entrée par le clavier
\$657A	16	Nom d'une variable ou d'un tableau
\$65AC	2	Avant dernière adresse de la RAM
\$65B1	1	Code de la dernière touche enfoncée au clavier

Annexe 10

Principales routines du BASIC

<i>Adresse hexa</i>	<i>Commentaire</i>	<i>Page</i>
2F3	Recherche dans la pile le FOR associé au NEXT rencontré	73
336	Test de dépassement de capacité mémoire (OM Error)	
353	Traitement de l'erreur dont code dans B	55
4A0	Recherche d'un numéro de ligne	
66B	Recherche de la fin de l'instruction courante	70
66E	Recherche de la fin de la ligne courante	70
6FD	Calcul d'une étiquette (numéro de ligne)	
734	Conversion et rangement d'une valeur (numérique ou chaîne)	62
770	Calcul de la valeur d'un opérande	64
7EB	Contrôle de syntaxe ; caractère suivant→A	63
800	Valeur (adresse si chaîne) d'une variable→Accu. flottant FAC	
81A	Calcul d'une expression, retournée dans FAC	63
8BD	Permutation des 2 accus. flottants (FAC en \$6155, et \$6163)	

<i>Adresse hexa</i>	<i>Commentaire</i>	<i>Page</i>
A48	Recherche (et création éventuelle) d'une variable ou d'un élément de tableau	60
B05	Calcul d'une valeur entière positive, retournée dans \$6157,58	
C59	Registre D→\$6157,58 ; #02→\$6105 (type entier)	
1055	Affichage du caractère dont code dans A	
16AC	Recherche du WEND associé au WHILE rencontré	74
16AD	Recherche du NEXT associé au FOR rencontré	74
1ABF	Chargement du 2d Accu. flottant (\$6163) avec une valeur réelle	
1C02	Chargement de FAC avec une valeur (adresse dans X)	
1C38	Inverse de 1C02 : FAC→mémoire	
1C64	2d accu. flottant→FAC	
1C81	Inverse de 1C64 : FAC→2d accu. flottant	
1CC8	Test de valeur logique " FAUX"(0) dans FAC (→Z=1)	
1ED1	Affichage de la valeur contenue dans D	
2502	Détermination du type d'une valeur (positionne CC) . . .	63
2510	Conversion en cas de mélange de type	
2AED	Boucle d'exécution des programmes	55
2B25	Traitement des instructions	53
34CB	Traitement des coordonnées d'un point graphique	
354B	Traitement des paramètres des instructions graphiques	76

Page 0 :

<i>Adresse hexa</i>	<i>Commentaire</i>	<i>Page</i>
61B2(\$B2)	Caractère courant suivant retourné dans A	53
61B8(\$B8)	Caractère courant retourné dans A	54
61CA(\$CA)	Vérifie que " , " ; caractère suivant	
61CD(\$CD)	Branche en \$2502 (détermination d'un type)	
61D0(\$D0)	Branche en \$7EB (contrôle de syntaxe)	

REMARQUES : A,B,CC et X désignent les registres du 6809

FAC désigne l'accumulateur flottant,situé en \$6155